CSC/CYEN 131

The Science of Computing II

Living *with* Cyber

Student Edition

CSC/CYEN 131: The Science of Computing II
Living *with* Cyber (part 2 of 3)

Course Description: Intermediate algorithm analysis and development, object-oriented programming, high-level data structures, computer architecture, and problem solving. This is the second Living *with* Cyber course.

Course Outcomes: Upon **successful completion** of this course, students should:

1. Be able to identify a problem's variables, constraints, and objectives;
2. Be able to write object- and non-object-oriented programs in a general-purpose programming language (e.g., Python);
3. Be able to transform numbers from base two to bases ten and sixteen – and vice versa;
4. Understand binary addition and multiplication;
5. Have exposure to some applications of computing (e.g., graphical user interfaces);
6. Have a basic understanding of randomness, probability, and pseudo-random number generators;
7. Have a basic understanding of recursion and breaking problems down; and
8. Have a basic understanding of high level data structures (e.g., linked lists, stacks, queues, binary trees).

Prerequisite(s): A grade of **C** or better in CSC 130 or CYEN 130.

Textbook: The Living *with* Cyber text (in PDF format) is available for free online at www.livingwithcyber.com.

Grades: Your grade for this class will be determined by dividing your total earned points by the total points possible. In general, graded components will fall into the following categories:

Attendance:	~2.5%
Puzzles:	~2.5%
Raspberry Pi activities:	~27.5%
Programs:	~17.5%
Major tests:	~50%

The Raspberry Pi kit that will be used throughout the Living *with* Cyber curriculum in the 2017-18 academic year will be provided to participating students at no cost. Students who drop the Living *with* Cyber curriculum before finishing it must return the kit. Students not majoring or minoring in Computer Science, or majoring Cyber Engineering, will be loaned the kit and must return it at the completion of the Living *with* Cyber curriculum. Please see www.livingwithcyber.com for more information about device requirements.

Students needing testing or classroom accommodations based on a disability are encouraged to discuss those needs with me as soon as possible. For more information, please visit www.latech.edu/ods.

If you are ill, you can get treatment at the Wellness Center in the Lambright Intramural Center building. The nurses there can treat minor illnesses and can give vouchers to see doctors in town for more serious illnesses. Since you have already paid for this service through your fees, there is usually no additional charge. Also, if you sign a HIPPA release form at the time of your visit, they can verify that you were ill and thus you will have an excused absence for missing class.

In accordance with the Academic Honor Code, students pledge the following: "Being a student of higher standards, I pledge to embody the principles of academic integrity." For the Academic Honor Code, please visit http://www.latech.edu/documents/honor-code.pdf.

All Louisiana Tech students are strongly encouraged to enroll and update their contact information in the Emergency Notification System. It takes just a few seconds to ensure you're able to receive important text and voice alerts in the event of a campus emergency. For more information on the Emergency Notification System, please visit http://ert.latech.edu.

TOPICS COVERED:

- More Python
- The Object-Oriented (OO) Paradigm
- Number Systems and Binary Arithmetic
- Application (Beam): Graphical User Interfaces
- Chaos
- Recursion
- High Level Data Structures

Lesson Summary

#	Title	Pillar(s)	Description/Topic(s)	Periods
01	More Python	Computer Programming	1. What you should already know 2. A review of program flow 3. Formal vs. actual parameters 4. Variable scope 5. A review of the Python list 6. Revisiting searching and sorting in Python 7. Exiting repetition constructs early 8. Other operators 9. String methods 10. Importing external libraries and designing modules	3
02	The Object-Oriented Paradigm	Computer Programming	1. Introduction to the object-oriented paradigm 2. State and behavior 3. Objects, classes, and instances 4. Class definitions 5. Object references 6. Instance variables vs. and class variables 7. Accessors and mutators 8. Range checking and input validation 9. Operator overloading 10. Class diagrams 11. Inheritance	6
04	Number Systems and Binary Arithmetic	Computer Architecture	1. The binary number system 2. The hexadecimal number system 3. Number system conversion 4. Binary arithmetic (addition and multiplication) 5. Binary addition (including half adders, full adders, and chaining full adders) 6. Bitwise operators in Python	3
05	Graphical User Interfaces	Beam (Application #1)	1. GUI components 2. Events 3. The Python Tkinter library 4. Common GUI widgets 5. Configuring and positioning widgets 1. Various GUI examples	1
06	Chaos	Algorithms	1. The coordinate system 2. The Chaos Game 3. Fractals, randomness, and probability 4. Random number generators	2
07	Recursion	Algorithms	1. The Towers of Hanoi	2

			2. Breaking problems down 3. Recurrence relations 4. Recursion 5. Famous recursive algorithms (e.g., factorial, Fibonacci) 6. The Towers of Hanoi...reloaded	
08	High Level Data Structures	Data Structures	1. A review of arrays 2. Linked lists 3. Stacks 4. Queues 5. Binary trees	2
Pi Activities				5
Exams				3
Housekeeping				1
Slack				2
TOTAL				**30**

CSC/CYEN 131: The Science of Computing II

Last updated:

13 Mar 2018

		Lessons	Raspberry Pi Activities	Puzzles	Videos	Assessments	
WEEK 1	1	Housekeeping		Alphabet Equation -- How Many Zeros?	Ira Glass on Storytelling		W
	2	More Python				Program 1 How Many Zeros?	F
	3						M
WEEK 2	4					Program 2 Sort Comparisons and Swaps	W
	5	The Object Oriented Paradigm					F
	6						M
WEEK 3	7					Program 3 A Simple Vehicle Class	W
	8			Oddball Sum -- AB + BA = CDC	Slomo		F
	9		Pi Activity 1 Room Adventure				M
WEEK 4	10	The Object Oriented Paradigm				Program 4 Fraction Enhance	W
	11						F
	12					Exam 1 More Python The Object Oriented Paradigm	M
WEEK 5	13	Number Systems and Binary Arithmetic					W
	14						F
	15			Mary's Siblings -- Jar of Marbles	Kara – Detroit: Become Human	Program 5 A Simple Vehicle Class...Reloaded	W

CSC/CYEN 131: The Science of Computing II

Last updated:

13 Mar 2018

		Lessons	Raspberry Pi Activities	Puzzles	Videos	Assessments	
WEEK 6	16		Pi Activity 2 My Binary Addiction				F
	17					Exam 2 Number Systems and Binary Arithmetic	M
	18	Beam Graphical User Interfaces					W
WEEK 7	19	Chaos					F
	20			Various Sequences -- The Look and Say Sequence	The Singularity	Program 6 A Foolproof Game	M
	21		Pi Activity 3 The Reckoner				W
WEEK 8	22	Recursion					F
	23						M
	24	High Level Data Structures		Birthday of Three Sons -- Three Switches and a Bulb		Program 7 Sort Comparisons and Swaps...Reloaded	W
WEEK 9	25						F
	26					Exam 3 Chaos Recursion High Level Data Structures	F
	27		Pi Activity 4 Simon				M
WEEK 10	28						W
	29	SLACK					F
	30						M

Lessons

What you should already know
In this lesson, we will build on what you have already learned about the Python programming language. To be sure that we are all on the same page, let's briefly review the things about Python that you should already be familiar with. For more detail, review the lesson on *Introduction to Computer Programming*.

Data types, constants, and variables
You should know that the kinds of values that can be expressed in a programming language are known as its **data types**. The **primitive types** of a programming language are those data types that are built-in (or standard) to the language and typically considered as basic building blocks (i.e., more complex types can be created from these primitive types). Python's standard types can be grouped into several classes: numeric types, sequences, sets, and mappings. You should be familiar with numeric types and sequences (e.g., lists).

You should know that a **constant** is defined as a value of a particular type that does not change over time. In Python both numbers and text may be expressed as constants. **Numeric constants** are composed of the digits 0 through 9 and, optionally, a negative sign (for negative numbers), and a decimal point (for floating point numbers). **Text constants** consists of a sequence of characters (also known as a string of characters – or just a **string**).

You should know that a **variable** is defined to be a named object that can store a value of a particular type. Before a variable can be used, its name must be declared.

Input and output
You should be familiar with obtaining input (via the `input` function) and generating output (via the `print` statement) in Python. Here's a simple example:

```
name = input("What is your name? ")
print "Hello, {}!".format(name)
```

Operators
You should be familiar with a variety of operators in Python. Specifically, arithmetic operators, relational (comparison) operators, and assignment operators. Arithmetic operators include addition, subtraction, etc, and perform arithmetic operations on operands. Relational operators include comparison of equality, inequality, less-than, and so on, and perform comparisons on operands and return true or false. Assignment operators include operators such as +=, -=, and so on, and combine assignment with arithmetic.

Primary control constructs
You should be very familiar with the three primary control constructs: sequence, selection, and repetition. Sequence implies one statement after another. Selection allows blocks of optional statements to be executed. Repetition provides a mechanism for repeating blocks of statements. There are two main forms of repetition that we have covered: iteration and recursion. Iteration involves repeating a task some fixed number of times, until a condition is reached, or over some structure (such as the items in a list). Although recursion was only briefly covered, you should know that it involves breaking a problem down repeatedly into smaller versions of itself until a base or trivial case is reached. We will cover recursion in much more detail later in the curriculum.

Subprograms
You should be quite familiar with subprograms and how they can encapsulate behavior in programs. They are organized, reusable, and related statements that perform some action. Specifically, some subprograms perform tasks and terminate; others return a value. You should understand how control flow is transferred to a subprogram when a subprogram is called, and how it is returned when the subprogram terminates.

A review of program flow
Although you should be familiar with this already, it is so important that we should probably go over it in detail again. It is very important to be able to identify the flow of control in any program, particularly to understand what is going on. In fact, this significantly helps to debug problems in programs. Recall that, in Python, function definitions aren't executed in the order that they are written in the source code. Functions are only executed when they are called. This is perhaps best illustrated with an example that you have seen before:

```
 1:  def min(a, b):
 2:       if (a < b):
 3:            return a
 4:       else:
 5:            return b

 6:  def max(a, b):
 7:       if (a > b):
 8:            return a
 9:       else:
10:            return b

11:  num1 = input("Enter a number: ")
12:  num2 = input("Enter another number: ")
13:  print "The smaller is {}.".format(min(num1, num2))
14:  print "The larger is {}.".format(max(num1, num2))
```

Each Python statement is numbered for reference. Lines 1 through 5 represent the definition of the function `min`. This function returns the *minimum* of two values provided as parameters. Lines 6 through 10 represent the definition of the function `max`. This function returns the *maximum* of two values provided as parameters. Lines 11 through 14 represent the main part of the program. Although the Python interpreter does *see* lines 1 through 10, those lines are not actually *executed* until the functions `min` and `max` are actually called. The first line of the program to actually be executed is line 11. In fact, here is the order of the statements executed in this program if *num1* = 34 and *num2* = 55:

11, 12, 13, 1, 2, 3, 14, 6, 7, 9, 10

Let's explain. Line 11 asks the user to provide some value for the first number (which is stored in the variable *num1*). Line 12 asks the user to provide some value for the second number (which is stored in the variable *num2*). Line 13 displays some text; however, part of the text must be obtained by first calling the function `min`. This transfers control to line 1 (where `min` is defined). The two actual parameters, *num1* and *num2*, are then passed in and mapped to the formal parameters defined in `min`, *a* and *b*. Then, line 2 is executed and performs a comparison of the two numbers. Since *a* = 34 and *b* =

55, then the condition in the if-statement is true. Therefore, line 3 is executed before control is transferred back to the main program with the value of the smaller number returned (and then control continues on to line 14). Note that lines 4 and 5 are never executed in this case!

Line 14 is then executed and displays some text. Again, part of the text must be obtained by first calling the function `max`. This transfers control to line 6 (where `max` is defined). The variables *a* and *b* take on the values 34 and 55 respectively. Line 7 is then executed, and the result of the comparison is false. Therefore, line 8 is not executed. Control then goes to line 9, and then to line 10 which returns the larger value. The program then ends.

What is the order of execution if *num1* = 55 and *num2* = 34?

In `min`, *a* > *b* (i.e., *num1* > *num2*); therefore, the else (false) part is executed (i.e., line 3 is never executed). Similarly, the true part of `max` is executed (i.e., lines 9 and 10 are never executed).

What if *num1* = 100 and *num2* = 100?

Since *a* == *b* (i.e., *a* is not greater than *b*, but *a* is also not less than *b*), the the else part of each function is executed. That is, lines 3 and 8 are never executed.

Here's another example with a simple for loop:

```
1:   for a in range(1, 4):
2:        for b in range(1, 5):
3:             print "{} * {} = {}".format(a, b, a * b)
```

This snippet of code displays a portion of a multiplication table. In fact, here's the output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
```

Here's the order of the statements executed. To make things a bit more clear, it is grouped and highlighted:

1, 2, 3, 2, 3, 2, 3, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 3

|——— 1 * n ———|——— 2 * n ———|——— 3 * n ———|

The first portion (highlighted in red and labeled 1 * n) represents a single iteration of the **outer** for loop and a *full* iteration of the **inner** for loop. It generates the following output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
```

Let's explain. Line 1 of the outer for loop generates the list [1, 2, 3]. It then iterates over the values in the list with the variable *a* taking on each value, one at a time. Initially, *a* = 1. Line 2 represents the inner for loop and generates the list [1, 2, 3, 4], and iterates over its values with the variable *b* taking on each value, one at a time. Initially, *b* = 1. Line 3 then displays the first line of output: 1 * 1 = 1. This makes sense because *a* and *b* are both 1.

So far, the order of statements executed is 1, 2, 3. Note that the inner for loop iterates its complete cycle (i.e., through the entire generated list) for each iteration of the outer for loop. Therefore, the inner for loop iterates through the list [1, 2, 3, 4] for each value in the outer for loop's list [1, 2, 3]. After line 3 (when *a* and *b* are both 1), the inner for loop then iterates to the next value in the list. Therefore, line 2 is executed again so that *b* = 2. Similarly, line 3 executes again, generating the output: 1 * 2 = 2. At this point, the order of statements executed is 1, 2, 3, 2, 3.

The inner for loop continues iterating two more times (lines 2 and 3), setting *b* to 3 and then to 4. After the first full iteration of the outer for loop, the order of statements executed is 1, 2, 3, 2, 3, 2, 3, 2, 3. So when *a* = 1, *b* goes through the values 1, 2, 3, and 4. The output generated at this point is then:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
```

Since the inner for loop has finished a full iteration, control goes back to line 1, thereby allowing the outer for loop to iterate to the next value in the list so that *a* = 2. Line 2 is executed again, generating a new list [1, 2, 3, 4] and setting *b* to 1. Similarly, line 3 is executed again, generating the output: 2 * 1 = 2. Lines 2 and 3 are executed as before, for each value in the inner loop's list [1, 2, 3, 4]. Of course, this generates exactly the same order of statements as before: 1, 2, 3, 2, 3, 2, 3, 2, 3. However, this represents the second iteration of the outer loop.

Since the inner for loop has finished another full iteration, control goes back to line 1, thereby allowing the outer for loop to iterate to the next value in the list so that *a* = 3. Again, lines 2 and 3 are executed as before, for each value in the inner loop's list [1, 2, 3, 4]. Clearly, this generates exactly the same order of statements as before: 1, 2, 3, 2, 3, 2, 3, 2, 3. This time, it represents the third (and final) iteration of the outer loop. Why? Because the outer for loop has iterated through the entire list [1, 2, 3]. After this final iteration, the variable *a* has taken on all of these values. Therefore, the outer for loop is exited, and the program terminates.

Again, knowing the order in which statements are executed is crucial to debugging programs and ultimately to creating programs that work.

This concludes a review of what you should already know in Python. From here, we'll introduce new content.

Formal vs actual parameters

You have seen that a function can have parameters. These parameters are formally stated when the function is defined; for example:

```
def average(a, b):
    return (a + b) / 2.0
```

Here, the variables *a* and *b* are formally defined as parameters that must be passed in to the function `average` when it is called. In this context, the variables *a* and *b* are called **formal parameters**. It is where they are defined (in a formal manner).

Now consider a point in the source code where this function is called; for example:

```
avg = average(11, 67)
```

Here, the result of a call to the function `average` with the supplied values (or parameters) 11 and 67 is stored in the variable *avg*. These values, 11 and 67, are considered **actual parameters** in this context. That is, they are the *actual* values that will be passed in as parameters to the function `average`. In fact, they are mapped to the formally defined parameters (i.e., formal parameters) *a* and *b* in the function `average`. That function will use these values to make calculations and return the average of the two. The value returned replaces the function call. Think of this replacement as follows:

```
avg = ~~average(11, 67)~~
          39.0
```

Therefore, the variable *avg* is assigned the value 39.0 after the call to the function `average` is complete. Consider this call to the same function:

```
x = 11
y = 67
avg = average(x, y)
```

Here, the result is still the same. The average of the two variables, *x* and *y* (with the values 11 and 67 respectively), is stored in the variable *avg*. Here, *x* and *y* are also actual parameters (even if they are variables themselves) because they represent the actual values supplied to the function `average`.

Variable scope

Consider the following Python program snippet:

```
a = 10

def f(x):
    a = 11
    b = 21
    x *= 2
    print "in f(): a={}, b={}, x={}".format(a, b, x)

b = 20
f(b)
print "in main: a={}, b={}".format(a, b)

def g():
    global a
```

```
        a *= 1.5
        print "in g(): a={}, b={}".format(a, b)

    g()
    print "in main: a={}, b={}".format(a, b)
```

The variables *a* and *b* (highlighted above) are considered **global variables**. That is, they are accessible throughout the entire program because they are defined outside of any block context (e.g., a loop construct, a function, etc). Global variables can be accessed anywhere. Their **scope** is global (i.e., throughout the entire program). Take a look at the output of the program above:

```
in f(): a=11, b=21, x=40
in main: a=10, b=20
in g(): a=15.0, b=20
in main: a=15.0, b=20
```

Let's explain the output. Initially, the variable *a* is assigned the value 10. The next segment of code defines the function `f`. This is only a definition (i.e., the statements are not actually interpreted or executed at this point). Then, the variable *b* is assigned the value 20. What follows is a call to the function `f`, passing the variable *b* as an **actual parameter**. Control is then transferred to the function `f`, whose statements are now executed. Note that, to the function `f`, the variable *x* is the formal parameter that takes on the value passed in (from the variable *b*). So the variable *x* is now equal to the value of the variable *b* (i.e., 20) that was passed in at the point of the call to `f`. Note that the variable *x* is local to the function `f`; therefore, it is considered a **local variable**. That is, it is defined in `f` and only accessible in `f` – its **scope** is valid only in the function `f`. Also note that, although *a* and *b* are global, there are **local** versions declared in `f`. It is important to note that these are different variables than the global versions – even if they have the same name!

So what happens in `f`? The local variable *a* is initialized with the value 11, the local variable *b* is initialized with the value 21, and the local variable *x* (which is passed in as an argument with the value 20) is doubled to 40. The output of the function `f` is then clear:

```
in f(): a=11, b=21, x=40
```

Once `f` completes and control is transferred back to the point at which function `f` was called, the variable *x* is no longer accessible! In fact, let's alter the print statement immediately after the call to `f` from:

```
print "in main: a={}, b={}".format(a, b)
```

And change it to:

```
print "in main: a={}, b={}, x={}".format(a, b, x)
```

Here's the output of the program now:

```
in f(): a=11, b=21, x=40
Traceback (most recent call last):
  File "scope.py", line 11, in <module>
    print "in main: a={}, b={}, x={}".format(a, b, x)
NameError: name 'x' is not defined
```

Notice the error indicating that the variable *x* is not defined. That's because it was defined in `f`; however, the current context is outside of `f`. The variable *x* is no longer available once `f` finishes and control is transferred back to the main part of the program.

Let's replace the print statement to remove the error and explain the rest of the output from the original execution of the program:

```
in main: a=10, b=20
in g(): a=15.0, b=20
in main: a=15.0, b=20
```

Once control is transferred back to the main part of the program, the local variables a and b (in f) no longer exist. However, the global variables a and b do! They were initialized to 10 and 20 respectively. Therefore, the next line of output makes sense:

```
in main: a=10, b=20
```

The next part of the program defines another function, `g`, that is then called. Note the **global** keyword in the function `g`. This instructs Python to reference a globally defined version of the variable that follows the global keyword. That is, a local version is not defined and/or initialized. Instead, the global version is directly referenced. Moreover (and quite importantly), it permits the global version to be **changed**. Although there are no arguments to the function `g`, the global variable *a* is directly **modifiable** through the global keyword. When the statement `a *= 1.5` is executed, the value of the global variable *a* is 10. This statement changes its value to 15.0, directly updating the variable's value – globally!

Note the print statement in `g`. It refers both to the variables *a* and *b*. A reference to the variable *a* makes sense; however, the variable *b* is also accessible. In fact, the variable *b* is referencing the global version of *b*, similar to the variable *a* (i.e., it is directly **readable**). The difference in using the global keyword is that it permits a change to the variable; without it, it can only be utilized in a read-only manner. Since the global variable *b* is initialized with the value 20, the output in `g` is clear:

```
in g(): a=15.0, b=20
```

When control is transferred back to the main part of the program, changes to global variable *a* persist (even if they were changed in a function!):

```
in main: a=15.0, b=20
```

To illustrate this even more, let's slightly change the function `g` as follows:

```
def g():
    global a
    a *= 1.5
    b = 40
    print "in g(): a={}, b={}".format(a, b)
```

Note the slight difference: *b* is declared and initialized with the value 40. Which *b* is this? Is it a local version (i.e., local to `g`)? Or is it referring to the global version declared in the main part of the program? Recall that, without the global keyword, a global variable can not be modified. Therefore, an assignment statement in a function to a variable that has the same name as a global variable indicates that the variable is a new instance, defined **locally** in the function. This is a different variable *b*! The output in `g` is clear:

```
in g(): a=15.0, b=40
```

When control is transferred back to the main part of the program, the local version of *b* disappears. All that's left is the global version (that remains unchanged at 20). Therefore, the output in the main part of the program is also clear:

```
in main: a=15.0, b=20
```

A review of the Python list

Although you should be familiar with Python lists, they are quite important and used often; therefore, we will go over it again. Generally, a Python **sequence** is composed of (typically related) elements. Each element in a sequence is assigned an index (or position). A sequence with *n* elements has indexes 0 to *n*–1. Python has many built-in types of sequences; however, the most popular is called the list.

The **list** in Python is quite versatile. Recall that a list is declared using square brackets; for example:

```
grades = [ 94, 78, 100, 86 ]
```

The statement above declares the list `grades` with four integers: 94, 78, 100, and 86. The list can be displayed in its entirety (e.g., with the statement `print grades`); however, we can access each element individually by its index (specified within brackets). Accessing can mean to read a value in the list, or it can mean to change a value in the list; for example:

```
print grades[0]
grades[3] = 87
grades[1] += 2
```

Recall that more than one value in a list can be accessed at a time. We can specify a range (or interval) of indexes in the format `[lower:upper+1]` which means the interval [lower, upper) (i.e., closed at lower and open at upper). That is, the lower index in the range is inclusive but the upper is not. For example:

```
stuff[3:4]      # accesses index 3 (the same as stuff[3])
stuff[0:5]      # accesses indexes 0 through 4
stuff[-3]       # accesses the third index from the right
```

Also recall that list elements can be deleted with the **`del`** keyword as follows:

```
del stuff[2]
```

Finally, recall that Python provides several built-in operations that can be performed on lists. Here are many of them:

`len(list)`	Returns the length of a list
`max(list)`	Returns the item in the list with the maximum value
`min(list)`	Returns the item in the list with the minimum value
`list.append(item)`	Inserts item at the end of the list
`list.count(item)`	Returns the number of times an item appears in the list
`list.index(item)`	Returns the index of the first occurrence of item
`list.insert(index, item)`	Inserts an item at the specified index in the list
`list.remove(item)`	Removes the first occurrence of item from the list

`list.reverse()`	Reverses the items in the list
`list.sort()`	Sorts a list

Revisiting searching and sorting in Python
In previous lessons, we designed several searching algorithms (sequential/linear search and binary search) and sorting algorithms (bubble sort, selection sort, and insertion sort). We first specified them in pseudocode, and for some we showed how they could be implemented in Python (sequential search, binary search, and selection sort). To help get a better understanding of Python, let's briefly revisit some of these.

First, here's the **sequential search** for the smallest value in a list (from an earlier lesson). Note that a Python list is first populated with 20 random integers (from 1 to 99, also from an earlier lesson):

```
 1:  from random import randint

 2:  numbers = []
 3:  while (len(numbers) < 20):
 4:       numbers.append(randint(1, 99))

 5:  print numbers

 6:  minIndex = 0

 7:  for index in range(1, len(numbers)):
 8:       if (numbers[index] < numbers[minIndex]):
 9:            minIndex = index

10:  print "The smallest value is at index: {}".format(minIndex)
11:  print "The smallest value is: {}".format(numbers[minIndex])
```

This version of the sequential search technically returns the **index** of the smallest value (which is typically what programmers are interested in). Since the value can be easily accessed through the index, returning the index is much more meaningful. To generalize the sequential search so that it can return the index of a *specified* value (as opposed to the smallest value), it can be modified by replacing lines 6 through 11 as follows:

```
num = input("What integer would you like to search for? ")

for index in range(len(numbers)):
     if (numbers[index] == num):
          print "The value {} was found at index {}!".format(num, index)
```

What happens if the specified value is duplicated several times in the list? Clearly, each index would be displayed. Here's example (with user input highlighted in red):

```
[20, 47, 80, 52, 98, 80, 1, 14, 31, 48, 70, 31, 97, 30, 31, 43, 59, 2, 38, 50]
What integer would you like to search for? 31
The value 31 was found at index 8!
The value 31 was found at index 11!
The value 31 was found at index 14!
```

But what if it's only necessary to find the *first* occurrence of a specified value (and then abort)? Python provides a way to **exit a repetition construct early** through the **`break`** keyword! Formally, the break keyword exits the *nearest enclosing* repetition construct. More on this in a bit. To illustrate the use of the break keyword, the sequential search code above can be modified to return only the first instance of a specified value:

```
for index in range(len(numbers)):
    if (numbers[index] == num):
        print "The value {} was found at index {}!".format(num, index)
        break
```

Here's an example:

```
[87, 44, 37, 69, 92, 74, 49, 97, 65, 69, 27, 61, 22, 77, 3, 3, 25, 86, 53, 45]
What integer would you like to search for? 3
The value 3 was found at index 14!
```

Note that the value 3 occurs twice in the list (at index 14 and index 15); however, only the first instance is reported to the user before the search terminates. The break statement exits the enclosing repetition construct: in this case, the for loop.

What if the break keyword is located in a repetition construct that is also located inside of another repetition construct? In this case, it will exit the inner repetition construct only. Here's an example:

```
for i in range(2):
    print "i={}".format(i),
    for j in range(5):
        print "j={}".format(j),
        if (j > 1):
            break
    print
```

Here's the output:

```
i=0 j=0 j=1 j=2
i=1 j=0 j=1 j=2
```

The outer for loops iterates *i* from 0 through 1. The inner for loop iterates *j* from 0 through 4. Moreover, the inner for loop exits early if *j* is greater than 1. Technically, the print statement in the inner for loop will display values of *j* that are less than or equal to 1. So why is a value of 2 for *j* displayed? When *j* is 2, the value is displayed, after which the if statement is executed (which breaks out of the inner for loop). The outer for loop continues (the lone print statement is there to add a line break in between increasing values of *i*), and *i* becomes 1. This occurs again until the outer for loop terminates (when *i* is 2).

Let's now take a look at the binary search that was also covered in an earlier lesson. Recall that it is a very efficient search that requires a list to be sorted. Here's the Python code that was developed in an earlier lesson:

```
num = input("What integer would you like to search for? ")
found = False
first = 0
last = len(numbers) - 1
```

```
while (first <= last and found != True):
     mid = (first + last) // 2
     if (num == numbers[mid]):
          found = True
     elif (num > numbers[mid]):
          first = mid + 1
     else:
          last = mid - 1

if (found):
     print "{} was found at index {}!".format(num, mid)
else:
     print "{} was not found.".format(num)
```

This version of the binary search keeps tracks of two boundaries (*first* and *last*) that identify the beginning and end indexes of the current portion of the list. Initially, *first* is 0 and *last* is *n*-1 (i.e., the entire list). If the middle value of the current portion of the list does not match the specified value, the appropriate half of the list is "discarded" by modifying either *first* (to discard the left half) or *last* (to discard the right half).

Recall that the binary search required a list to be sorted, thereby taking advantage of the algorithm's efficiency improvement over the sequential search. Here's the **selection sort** that was developed in an earlier lesson:

```
n = len(numbers)

for i in range(0, n-1):
     minPosition = i

     for j in range(i+1, n):
          if (numbers[j] < numbers[minPosition]):
               minPosition = j

     temp = numbers[i]
     numbers[i] = numbers[minPosition]
     numbers[minPosition] = temp
```

Recall how the selection sort works: (1) the list is sorted from left to right; (2) at each pass (controlled by the outer for loop), the smallest value is swapped with the first item in the unsorted portion of the list; and (3) the inner for loop performs the comparison of every remaining item in the unsorted portion of the list to find the smallest value. For a review, see the lesson on *Searching and Sorting*.

There were two other sorting algorithms that were covered in earlier lessons: bubble sort and insertion sort. We never developed Python code for them. Let's do this now. First, the bubble sort. Here is a version in pseudocode:

```
for i ← 1..list length-1
     for j ← 1..list length-i
          if item j of list < item j-1 of list
          then
```

```
                    temp ← item j of list
                    item j of list ← item j-1 of list
                    item j-1 of list ← temp
               end
          next
     next
```

You may not have seen a for loop described in pseudocode before; however, this is a common way to accomplish this repetition construct in pseudocode. So what's happening here? The basic idea is that a value in the list will be compared to the one before it. If they are out of order, then they are swapped. This continues, one index over (to the right), until the end of the list is reached. After the first pass, the largest value is guaranteed to be in its final position (i.e., at the end of the list). The next pass starts again at the beginning of the list; however, this time comparisons and swaps only take place until the second-to-last value in the list (because the last value has already been placed there during the last pass). Each time, the sorted list grows from right-to-left until the entire list is sorted.

The outer for loop controls the number of passes, while also providing a way to reduce the size of the unsorted portion of the list after each pass. It iterates from 1 through *n*-1. The inner loop controls the comparisons and swaps. Initially, the inner loop begins at 1 (the index of the second value in the list), and compares this value to the one before it (the first value in the list). If they are out of order, they are swapped. The swap works by using a variable (*temp*) that temporarily takes on one of the values in the list. This continues with the next index (i.e., 2), and so on. The last index compared is *n*-*i*. If the algorithms is in the first pass (i.e., *i* is 1) and the length of the list is 10, the last valid index in the list is 10 – 1 = 9.

Let's take a look at what a Python version of the bubble sort looks like:

```
n = len(list)

for i in range(1, n):
     for j in range(1, n-i+1):
          if (list[j] < list[j-1]):
               temp = list[j]
               list[j] = list[j-1]
               list[j-1] = temp
```

Recall that Python's range function uses the first parameter as a lower bound and the second parameter as one above the upper bound. That is, it operates on the interval [*a*, *b*), where *a* is the (closed) lower bound and *b* is the (open) upper bound. Therefore, the upper bound of the outer loop is *n*: it iterates from 1 through (and including) *n*-1 as intended. Similarly, the upper bound of the inner loop is *n*-*i*+1: it iterates from 1 through (and including) *n*-*i* as intended. In the inner loop, if any value at an index is less than the value of the one before it, they are swapped.

Next, let's take a look at the Python code for the insertion sort. Recall that the insertion sort works somewhat as you would arrange a hand of cards being dealt to you: a new card is inserted in its appropriate position in the hand of cards dealt so far.

```
1:  i = 1

2:  while (i < n):
```

```
 3:        if (list[i-1] > list[i]):
 4:             temp = list[i]
 5:             j = i - 1

 6:             while (j >= 0 and list[j] > temp):
 7:                  list[j+1] = list[j]
 8:                  j -= 1

 9:             list[j+1] = temp

10:        i += 1
```

Here's an explanation of the code. Line 2 controls the number of passes through the list (*n*-1 total passes). The variable *i* is initialized to 1 (the second index in the list) and iterates through (but not including) *n*; therefore, through the last index in the list. So, starting with the second value in the list, it looks to the left (of this current value). Line 3 checks if that value is greater, and if so, then it must move it to the right. Line 4 temporarily stores the current item, and lines 5 and 6 then begin the process of iterating from the previous element, continuing to the left. At any point, if a value to the left is greater than the current item, it is shifted one index to the right. This continues either until (1) the beginning of the list is reached; or (2) a value that is not greater is found. Ultimately, the current item is placed into its proper position in the list. The outer loop then continues with the next value in the list (through the last value in the list).

Note that there are many other ways that the searches and sorts shown could have been implemented in Python. For example, the use of for loops in the selection sort could have been replaced with while loops (or vice versa in the insertion sort).

Other operators

Python provides several more classes of operators than you are already familiar with. Recall that, so far, you have been exposed to (and should be quite familiar with) arithmetic operators, relational (comparison) operators, and assignment operators. In this lesson, we will cover several other classes of operators: logical operators and membership operators.

The **logical operators** evaluate two operands and return the logical result (i.e., True or False). Think back to the primitive logic gates (*and*, *or*, and *not*). It turns out that they can be effectively mapped to conditions in if-statements. Logical operators operate on conditions (that use relational operators) and provide the overall logical result. In the following table, assume that *a* = True and *b* = False:

Python Logical Operators and Examples		
and	logical *and*	a and b is False
or	logical *or*	a or b is True
not	logical *not*	not a is False; not b is True

Note that this is equivalent to the primitive logic gates, where 0 is substituted for False and 1 for True. Here is the truth table for the and gate shown in this manner:

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

The logical operators sometimes make more sense when they are used in the context of a condition (e.g., in an if statement). Suppose that *a* = 5 and *b* = 10. The following condition would evaluate to False:

```
if (a == 5 and b < 10):
    ...
```

Why? Clearly because, although *a* is equal to 5, *b* is not less than 10 (it's equal to 10). Therefore, the *and* logical operator will return False if and only if both sides of the operator evaluate to True. In this case, the left side does while the right side does not. However, the following condition would evaluate to True:

```
if (a == 5 or b < 10):
    ...
```

The *or* logical operator will return True if either (or both) sides of the operator evaluate to True. Since *a* is equal to 5, then the left side is True. In this case, the right side doesn't need to be evaluated (and, in fact, it isn't – more on that below).

The logical operators do work when the inputs (i.e., *a* and *b* in the previous examples) aren't necessarily equal to True and False. That is, they also work when they are numeric values. Take, for example, the following statements:

```
1:   a = 23
2:   b = 13
3:   print a and b
4:   print a or b
5:   print not a
6:   print not b
7:   a = 0
8:   print not a
```

Here's the output (with lines numbers matching those of the print statements above):

```
3:   13
4:   23
5:   False
6:   False
8:   True
```

The output of lines 3 and 4 can be a bit confusing. Why, for example, is *a* **and** *b* 13? Or why is *a* **or** *b* 23? This can be explained by the following table, where the variables *a* and *b* have numeric values (as in the examples above):

Python Logical Operators and Examples

and	logical *and*	returns a if a is False, b otherwise
or	logical *or*	returns b if a is False, a otherwise
not	logical *not*	returns False if a is True, True otherwise

The output of lines 5, 6, and 8 makes sense when we realize that, in Python, 0 is False and any non-zero value is True! When *a* is 23 and *b* is 13, *a* evaluates to True (since it is non-zero); therefore, **not** *a* evaluates to False. This is the same with *b*. However, when *a* is 0, then it evaluates to False; therefore, **not** *a* evaluates to True. Formally, in the context of Boolean expressions, the following values are interpreted as false: False, None, numeric zero of all types, and empty strings and containers. All other values are interpreted as true.

Did you know?

The *and* and *or* logical operators are **short circuit** operators. That is, to evaluate a True or False result, the minimum number of inputs required to produce such an output is evaluated. For example, suppose that *a* = False and *b* = True. The expression *a* **and** *b* is only True if both *a* and *b* are True. Since *a* is False, then there is no need to evaluate (or test) the value of *b*. This would be useless and waste CPU cycles. Similarly, if *a* = True and *b* = True, the evaluation of the expression *a* **or** *b* only requires checking that *a* is True for the entire expression to evaluate to True (i.e., there is no need to evaluate/test the value of *b*).

Membership operators test for some value's membership in a sequence (e.g., to test if an element exists in a list, or if a character exists in a string). In the following table, suppose that the Python list *numbers* = [1, 3, 5, 7, 9], *x* = 2, and *y* = 3.

Python Membership Operators and Examples		
in	Returns True if a specified value is in a specified sequence or False otherwise	x in numbers is False; y in numbers is True
not in	Returns True if a specified value is not in a specified sequence or False otherwise	x not in numbers is True; y not in numbers is False

You have seen this in previous for loop examples (e.g., **`for`** *`i`* **`in`** *`list`*). This for loop configuration has the variable *i* take on each of the values in *list*, one at a time.

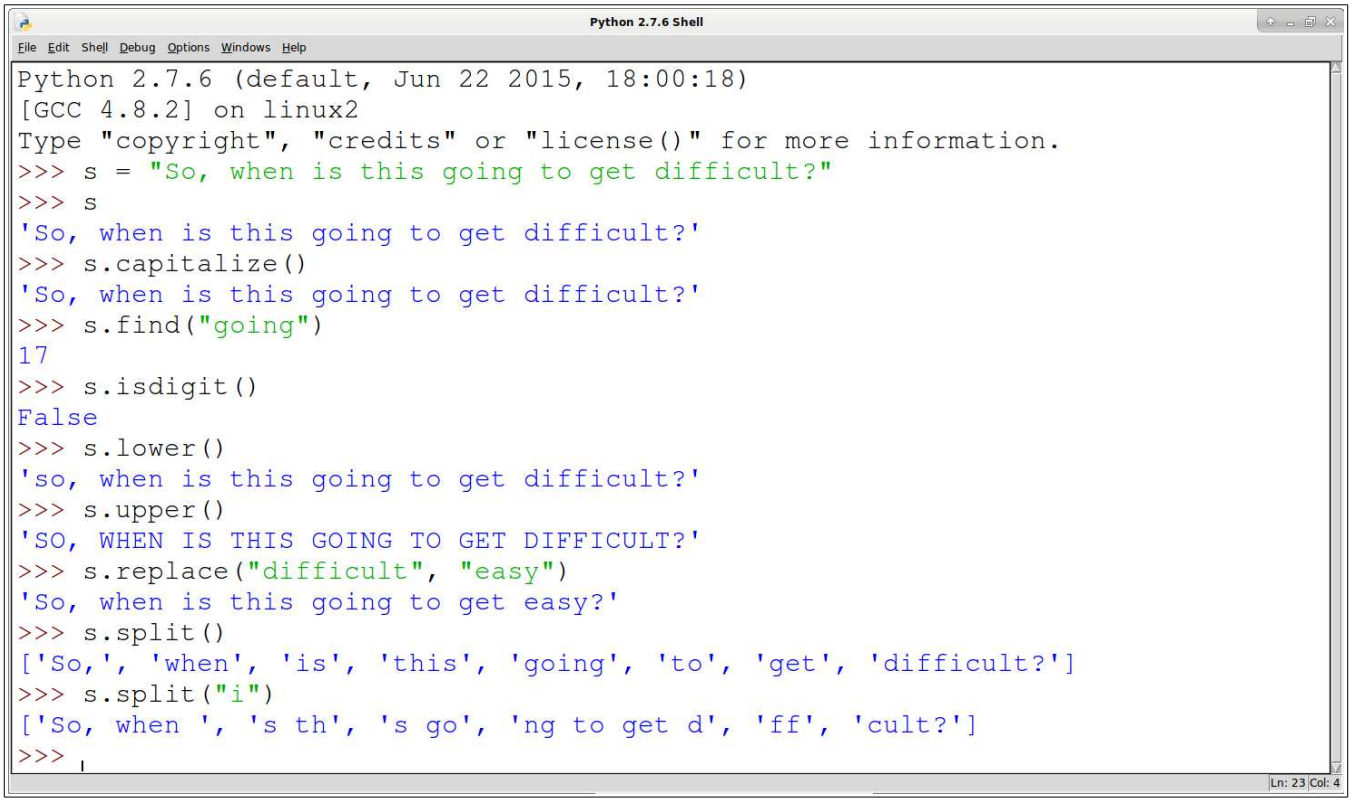
String methods

Strings are often necessary when writing programs. As such, Python provides a variety of methods that work on strings. You have already seen one such method, **format()**, that formats a string as specified (we did this earlier in one variant of the **print** statement). The following table lists some of the more useful string methods:

Python String Methods/Functions	
`str.capitalize()`	capitalizes the first character of a string
`str.find()`	returns the first index of a string within another string

`str.format()`	formats a string according to a specification
`str.isdigit()`	determines if a string consists only of numeric characters
`str.lower()`	converts a string to lowercase
`str.replace()`	replaces all occurrences of a string (within a string) with another string
`str.split()`	returns a list of the words in a string
`str.upper()`	converts a string to uppercase

These string methods are explained in greater detail in a variety of online sources. We suggest that you Google them and try them out. However, here are a few examples in IDLE:



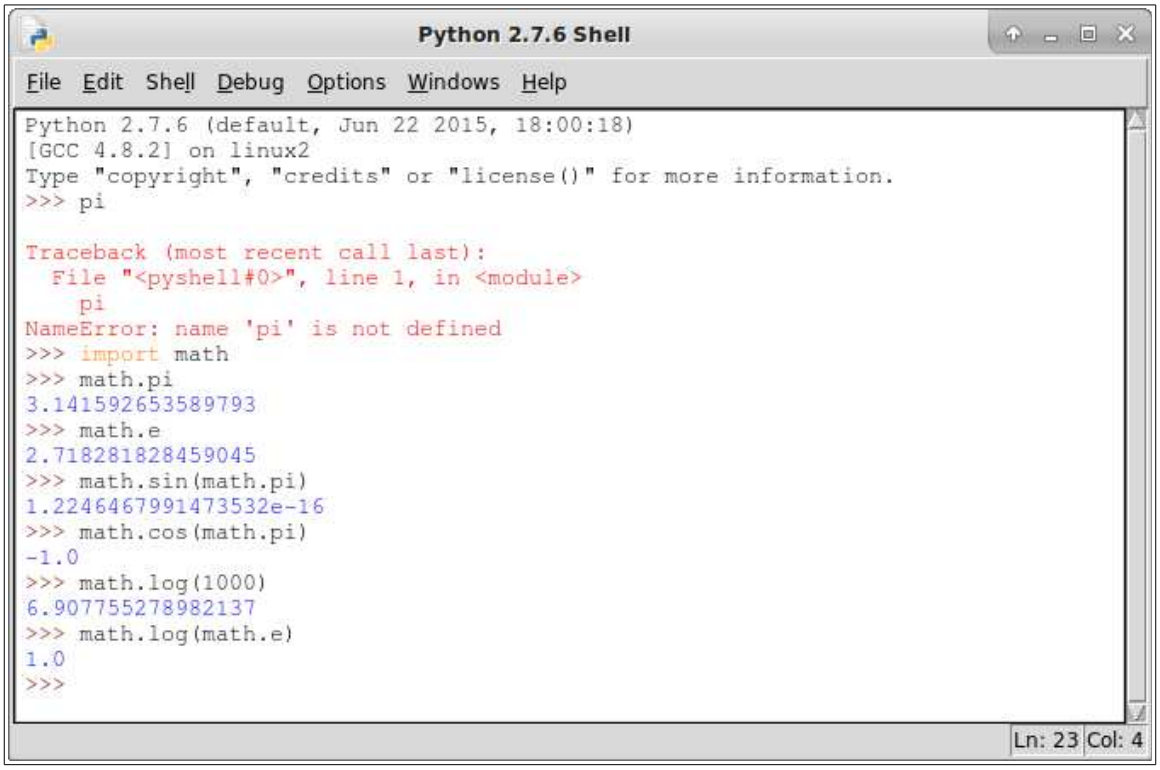
Note the execution of the string method **str.find()** above: `s.find("going")`. This string method returns the first index of the string, "going", within the string, *s*. Why is the result 17? At first glance, it seems that the first character of the string, "going", is at position 18. However, strings are sequences (just like lists); therefore, the characters of a string in Python begin at index 0.

Importing external libraries

It is often useful (and necessary) to import external functionality into our programs. In fact, you've seen (and used) this before (in the lesson *Introduction to Data Structures*), although it may not have been explained in detail. Often, others have designed functions and other bits of code that may prove useful. We don't always want to recreate things that already exist. Python supports the importing of such things via the **import** reserved word. For example, many of the programs we create require the use of mathematical functions beyond simple arithmetic (e.g., sin, cos, tan) or mathematical constants (e.g., pi, e). The structure of an import statement is as follows:

```
import library
```

Pretty simple. Here's an example of the importing and use of the math library:



Note in the example, the invalid use of *pi* before importing that math library. In addition, any value or function used in a library must be fully qualified with the name of the library (e.g., we need to specify *math.pi* and not just *pi*). Alternatively, we can itemize what we wish to import from a library. This allows us to use values and functions directly without having to specify the library name. The structure of such an import statement is as follows:

```
from library import function (or constant)
```

For example, the constant PI and the sin() function can be formally imported as follows:

```
from math import pi, sin
```

Moreover, these can be directly used as follows:

```
print pi
print sin(pi)
```

Here's the output:

```
3.141592653589793
1.2246467991473532e-16
```

Formally, Python calls its libraries **modules**. And we can even write our own modules! They are just Python programs that typically provide definitions of constants and functions that other Python programs import and make use of. Python modules just need to be saved as a .py file and located in the

same folder/directory as a Python program that needs to make use of it. For example, we could include several useful functions in a file called MyGoodies.py. Suppose that it contained the following:

```
from time import time

# starts a timer
def start_time():
     global start

     start = time()

# stops the timer and returns the time elapsed
def stop_time():
     stop = time()
     elapsed = stop - start

     return elapsed
```

The purpose of this example module is to use it to time how long algorithms take to execute. It's quite simple. The function *start_time* effectively starts a timer (through the time library's *time* function) by capturing the current "time" – which is essentially the number of seconds elapsed since an epoch defined in your operating system. For Unix and Unix-like operating systems (e.g., the "flavor" of Linux used on the Raspberry Pi), the epoch is 1970-01-01 00:00:00. The function *stop_time* captures the current time again (this time, after the algorithm has finished), and calculates and returns the difference between the two.

We can make use of this module as follows:

```
from MyGoodies import start_time, stop_time

# start a timer
start_time()

# do something that takes a little time
for i in range(100000000):
     pass

# stop the timer
duration = stop_time()

# display how long it took
print "Algorithm took {} seconds.".format(duration)
```

Note that the "algorithm" in the test code above really does nothing. It's just there to take up some noticeable amount of time so that the module can be tested. Here's a sample run:

```
jgourd@pi:~$ python MyGoodiesTest.py
Algorithm took 5.71634602547 seconds.
```

The Object-Oriented (OO) Paradigm Pillar: Computer Programming

As discussed in lessons early in the curriculum, three paradigms of programming languages have emerged over the years: the imperative paradigm, the functional paradigm, and the logical paradigm. A language is classified as belonging to a particular paradigm based on the programming features it supports. In addition, during the past decade or so these paradigms have been extended to include object-oriented features. Some computer scientists view object-oriented programming as a fourth paradigm. Others prefer to view it as an extension to the imperative, functional, and logical paradigms, in that object-oriented constructs and behaviors are often viewed as higher-level organizational attributes that can be incorporated into each of the three basic paradigms, rather than as a separate programming paradigm unto itself.

Object-oriented concepts have revolutionized programming languages. The vast majority of widely used programming languages are now object-oriented. In fact, they are, by far, the most popular type of programming languages. Python, Java, and C++ are object-oriented, imperative languages. Specifically, Python is an imperative, interpreted language that can *optionally* be object-oriented. That is, non-object-oriented programs can be written in Python if desired. In a sense, this makes Python quite powerful (like a Swiss army knife of programming).

The **object-oriented paradigm** is an elegant and clean way to conceptually think about computer programming. When used properly, it can produce programs that are more robust, less likely to have errors, and are easy for others to understand and modify. Specifically, the object-oriented approach adds the concepts of objects and messages to the paradigms listed above. We don't think of programs as procedures or lists of instructions to be executed in order from beginning to end; rather, we think of them as modeling a collection of objects interacting with each other.

Essentially, programs and the data on which they act are viewed as objects. In order to perform a task, an object must receive a message indicating that work needs to be done. Object-oriented languages are extremely useful for writing complex programs; for example, programs that support mouse-based, graphical user interfaces. Object-oriented programming helps in the construction of software systems by enabling large, complex systems to be subdivided into isolated functional units with well-defined external interfaces to other system components. There are many other distinguishing characteristics of object-oriented programs, including inheritance, polymorphism, and data encapsulation. Some of these will be discussed in this lesson, while others will be covered in later lessons.

State and behavior...the basic idea

We live in a world in which objects exist all around us. In fact, we interact with objects all the time. For example, the author of this document interacted with a keyboard and mouse while writing this sentence! The author's brain (an object) somehow sent a message to hands and fingers (all objects) to make contact with keyboard keys (again, all objects). The keyboard (an object) somehow sent a message to the computer (an object made up of many other objects) to interpret key presses as characters to be placed in this document (an object) and displayed on the screen (yet another object).

Fundamentally, an object is a thing. In object-oriented programming, the objects model things in our problem domain. Objects have properties (or attributes) that, in a sense, define them. They capture the properties or characteristics of an object. For example, a *person* object has many attributes, some of

which include sex, name, age, height, hair color, eye color, and so on. These are the things that a person can *be*. The collection of attributes that make up an object are called its **state**.

Objects can also *do* things. That is, they can have behaviors. In object-oriented programming, these behaviors are implemented by program modules (e.g., methods, procedures, functions, etc) that contain the instructions required to model the behavior in computer software. For example, a *person* object can eat, sleep, talk, walk, and so on. The collection of actions that an object can do are called its **behavior**.

Collectively, state and behavior define an object. When you begin to adopt this way of thinking, you can begin to see many things in our world as objects with attributes and behaviors interacting with other objects.

Objects, classes, and instances

An **object** represents a specific thing from the real world with defined attributes. For example, "the white truck down there parked by the road" is an object. It is a truck that could ostensibly be observed on the road. In fact, it could be a white 2016 4x4 Dodge Ram 1500 with 450 miles on it.

Clearly, there exist other trucks in the world. In fact, there may even be other trucks parked by the road next to the one just described. One could say, then, that the generic term *truck* could represent all kinds of truck objects. Trucks are all basically different versions of the same thing. That is, they all behave the same and have the same set of attributes; however, the values of those attributes is what sets them apart. For example, one truck could be red and another white; one truck could be a Dodge and another a Toyota.

A **class** represents a blueprint or template for those real world things with defined attributes (i.e., for all of the objects that can be derived). For example, a truck class could be used to create many truck objects. Another way of saying this is that a class is a collection of objects that share the same attributes and behaviors. The differences between individual objects are abstracted away and ignored. So the class can be thought of as the perfect idea of something. This occurs in the real world in, for example, the way a child learns to abstract away the differences between Aunt Jamie's schnauzer, a best friend's bulldog, and dad's boxer – and learns to classify them all as dogs.

This is not a new idea. Plato, quoting Socrates in *The Republic*, discusses the *Theory of Forms or Ideas*. For example, no one has ever seen a perfect circle; however, we have an idea of what a perfect circle should be. We have drawn many circles, but none of them were absolutely perfect. The perfect idea of a circle would be considered a class, and each of the circles we draw would be considered objects.

Formally, a class defines the state and behavior of a *class* of objects. The fact that a truck has a color, year, make, model, mileage, and so on, is defined in the class. The fact that a truck can haul, drive, turn, honk, and so on, is also defined in the class. In fact, how a truck hauls, drives, turns, and honks is specified in the truck class as well. From the truck class, many truck **instances** can be created, each with potentially different attribute values making up each truck's unique state. We say that, from this class, we can **instantiate** many objects. Usually, we use the term object and instance interchangeably. That is, a truck object, for example, is just an instance of the truck class.

Activity 1: The zoo

In this activity, you will play a game using an animal class. This class, formally called **Animal**, will define what animals can be and do. Some of the students in the class will become objects of the class **Animal**. The animal class defines several attributes that an animal has:

- `type`: a string that represents the animal's type (e.g., dog)
- `appetite`: an integer that represents how much daily food units the animal requires to live
- `stomach`: an integer that represents how much food units are currently in the animal's stomach
- `alive`: a Boolean that represents whether or not the animal is alive
- `sound`: a sound that represents the sound the animal makes

The class also defines several behaviors that an animal can do (and that students will perform when called upon):

- `talk()`: make the sound the animal makes
- `burn()`: use the animal's daily food units by subtracting `appetite` from `stomach`
- `eat(amount)`: increase the animal's `stomach` food units by the provided `amount`
- `getType()`: tell the requester what the animal's type is (i.e., the value of `type`)
- `isAlive()`: tell the requester if the animal is alive or not (i.e., the value of `alive`)

Note that if `stomach` is less than 0, then `alive` becomes false – and the animal dies...

Representing state and behavior

Objects store their state in **instance variables**. For example, a truck class could define the variable `year` to represent the year a truck was manufactured. A specific truck object (or instance) would set this variable to the year of its manufacture. In Python, this would be done with a simple assignment statement. If, for example, the truck object were manufactured in 2016, then the statement `year = 2016` would appropriately set the truck's year of manufacture. Another truck object could have a different year of manufacture. Ultimately, the class defines instance variables; however, each object stores its own unique set of values.

There are certain attributes that all instances of a class may want to share. Consider a class that defines a *person*. Although each instance/object of the person class can be different and thus have unique values stored in its instance variables (e.g., different sex, name, age, etc), all persons are Homo sapiens. All persons share this *scientific name*. In fact, if an expert in the field were to rename (or perhaps refine) the term Homo sapiens to something else, this would change for all persons, effectively at the same time. This kind of behavior can also be replicated in the object-oriented paradigm.

A **class variable** defines a value that is shared among all the instances of a class. Unlike instance variables that, when changed, only affect a single object, a change in a class variable affects all instances of a class simultaneously. Essentially, a class variable is stored in memory that is shared among all the instances of a class.

The behavior of objects is defined in **methods** (or functions) that can be invoked. For example, the *turn* behavior of a truck could be defined in a function called `turn`. If necessary, this function could take parameters as input and return some sort of output.

Ultimately, a class has source code that specifies its state (through instance variables) and behavior (through methods). Collectively, state and behavior are referred to as the **members** of a class. Let's take

a look at a simple example of a *dog* class in Python. For this example, a dog only has a name, and all dogs are canines.

```
1:   class Dog:
2:         kind = "canine"

3:         def __init__(self, dog_name):
4:              self.name = dog_name
```

Line 1 represents the class header, which includes the Python keyword `class` and the name of the class (in this case, *Dog*). Class headers are terminated with a colon, much like function headers. It is typical to capitalize the names of classes. Moreover, class names should always be singular nouns since they define the blueprint for a single thing.

Line 2 defines a class variable named `kind` that is initialized with the string "canine". This value is shared among all dogs. The reason that we consider `kind` to be a class variable is that it is defined inside the class but outside any methods that are in the class. That is, class variables are defined at the class level.

Lines 3 and 4 represent a function called `__init__`. In Python, functions that begin and end with two underscores have special meaning. In fact, they are called **magic functions**. The `__init__` function provides an initialization procedure for each instance of the class. That is, the source code contained within this method effectively defines what it means to initialize a new instance of the class. When we want new instances of the dog class, this function is automatically invoked. Formally, this type of function is called a **constructor** because it contains the source code required to *construct* a new instance of the class. A constructor is automatically invoked each time we create an instance of a class. Its purpose is to initialize an object, which typically means to set default values for one or more of the instance variables.

In the dog class above, the constructor takes two parameters: `self` and `dog_name`. The first parameter represents the instance that is about to be instantiated. **This parameter is always required!** The second parameter represents the name of this new dog (e.g., Bosco). The function header indicates that, to create a new instance of the dog class, a dog name must be provided. Line 4 then sets the instance variable `name` for the object to be created. Note that `self.name` (on the left side of the assignment statement) represents the instance variable (called `name`) for the dog class, and specifically targets this dog instance's name (via `self.name`). The *dot* in between `self` and `name` is called the dot operator and will be covered shortly. The variable `dog_name` (on the right side of the assignment statement) is passed in to the function when a new instance of a dog is desired.

Instances of the dog class can be created as we need them. This typically occurs outside of the dog class (for example, in a program that requires dog objects to interact with each other). Objects that are instances of the dog class can be easily instantiated as follows:

```
5:   d1 = Dog("Maya")
6:   d2 = Dog("Biff")
```

Line 5 declares a variable, `d1`, that represents an instance of the dog class. Specifically, `d1` is a dog whose name is "Maya". Line 6 defines a variable, `d2`, that represents another instance of the dog class. Specifically, `d2` is a dog whose name is "Biff".

When line 5 is executed, the variable `d1` is mapped to the variable `self` in the `__init__` function (constructor) of the dog class. The variable `self` is a formal parameter. The variable `d1` is the actual parameter that is mapped to the formal parameter. Similarly, the string "Maya" (actual parameter) is mapped to the variable `dog_name` (formal parameter). The statement `self.name = dog_name` ultimately sets the instance variable `name` for this instance of the dog class to whatever was passed in as the variable `dog_name` (i.e., Maya in this case).

Formally, the variable `d1` is called an **object reference**. That is, it refers to an object (or instance) of the dog class. The variable `d2` is also an object reference of the dog class. We can access the members of a class by using the **dot operator**. For example, we could change the name of `d1` to Bosco as follows:

```
d1.name = "Bosco"
```

The above example shows how to modify an instance variable. Note that it only changes the name of `d1` and not `d2` because the specified object reference is `d1`. Simply accessing (without changing) a member of a class is just as easy; for example:

```
print d2.name
```

This statement would produce the output Biff (because `d2`'s name is Biff). In fact, let's list the state of each instance of the class dog, `d1` and `d2` (note that this is not Python source code; rather, it is an enumeration of the instance variables and their associated values for the objects `d1` and `d2`):

`d1.name` → Bosco
`d2.name` → Biff
`d1.kind` → canine
`d2.kind` → canine

You have actually seen (and used) the dot operator before. Consider the following statements:

```
name = "Joe"
welcome_string = "Hello, {}"
print welcome_string.format(name)
```

From these statements, we can infer that strings (specifically the string `welcome_string` in the example above) are objects! In addition, the function `format` must be a member of the string class since it can be accessed using the dot operator! In fact, the function `format` is part of the behavior of the string class. This function takes one or more parameters that replace the empty braces in the string.

Did you know?

In Python, instance variables don't need to be formally declared in the class. That is, they can be defined as needed, dynamically. For example, although the dog class doesn't yet specify an instance variable that defines a dog's breed, the following statement in the main part of the program effectively adds the instance variable `breed` to the dog class. Specifically, it sets `d1`'s breed to German Shepherd:

```
d1.breed = "German Shepherd"
```

It is standard practice, however, to formally define all instance variables in the class. This will be further discussed later.

Instance variables vs. class variables

Suppose that an expert in the field decided to change the scientific name for dogs from canine to something like "caten". You know, to account for inflation[1]. Changing class variables separately for each instance of the dog class doesn't make sense. The purpose of a class variable is that its value is shared simultaneously among all of the instances of the class. The proper method of changing a class variable so that it appropriately affects all of the instances is to use the class name as follows:

```
Dog.kind = "caten"
```

This statement simultaneously changes the class variable `kind` (to caten) for all instances of the dog class. To illustrate this, here is some source code for a dog class that illustrates the behavior and differences of class and instance variables:

```
class Dog:
     kind = "canine"

     def __init__(self, dog_name):
          self.name = dog_name

d1 = Dog("Maya")
d2 = Dog("Biff")

print "I have a dog named {} that is a {} and another named {}\
       that is also a {}.".format(d1.name, d1.kind, d2.name,\
       d2.kind)
d1.name = "Bosco"
print "I have a dog named {} that is a {} and another named {}\
       that is also a {}.".format(d1.name, d1.kind, d2.name,\
       d2.kind)

Dog.kind = "caten"
print "I have a dog named {} that is a {} and another named {}\
       that is also a {}.".format(d1.name, d1.kind, d2.name,\
       d2.kind)

d1.breed = "German Shepherd"
d2.breed = "mutt"
print "I have a dog named {} that is a {} and another named {}\
       that is a {}.".format(d1.name, d1.breed, d2.name,\
       d2.breed)
```

Did you know?

You may have noticed above that some statements seem to be spread across multiple lines. Each of the lines that make up these statements end with a backslash (\), except for the last line of the statement. Python allows the use of a backslash to note that the remainder of a statement is provided on the next line. For example, take a look at the following statement:

[1] See Victor Borge's Inflationary Language (Google it!) for the meaning behind this.

```
    a = 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 -
1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1
```

This statement can be spread across multiple lines as follows:

```
    a = 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1\
          - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1 - 1 + 1
```

Here is the output of the program:

```
I have a dog named Maya that is a canine and another named Biff
 that is also a canine.
I have a dog named Bosco that is a canine and another named Biff
 that is also a canine.
I have a dog named Bosco that is a caten and another named Biff
 that is also a caten.
I have a dog named Bosco that is a German Shepherd and another
 named Biff that is a mutt.
```

Note a few things: (1) the class variable `kind` is applied to both instances, `d1` and `d2`; and (2) the instance variable `breed` is dynamically created and applied (separately) to each instance.

Did you know?

For readability, Python source code is presented as it is formatted in IDLE throughout this lesson. the main reason for this is that presenting source code this way provides syntax highlighting. **Syntax highlighting** is the feature of highlighting (or coloring) certain portions of source code so that it helps to categorize constructs, keywords, variables, and so on. It essentially helps to make the source code more readable. For example, Python keywords are colored orange and strings are colored green.

It is important to understand the difference between instance variables and class variables. Although they seem similar, they are actually quite different. Perhaps this is best illustrated with an example. Consider the following modified dog class:

```
class Dog:
     kind = "canine"
     tricks = []

     def __init__(self, dog_name):
          self.name = dog_name

     def add_trick(self, trick):
          self.tricks.append(trick)

d1 = Dog("Maya")
d2 = Dog("Biff")
d1.add_trick("roll over")
d2.add_trick("play dead")

print "I have a dog named {} that can {}.".format(d1.name,\
       d1.tricks)
```

```
print "I have a dog named {} that can {}.".format(d2.name,\
       d2.tricks)
```

The only difference in the class is the addition of the list `tricks` and the function `add_trick`. After all, a dog can do tricks! Adding a trick to an instance of the dog class can be done by accessing the `add_trick` function (using the dot operator) on an object reference of a dog instance and providing the trick to add (as a string). As shown before, the dog instance is automatically passed in and mapped to the formal parameter `self` in the function. The string that represents the trick to add is passed in and mapped to the formal parameter `trick`. The function appends a new trick to the end of the list.

The expected behavior of the source code above may be that each instance of the dog class (i.e., `d1` and `d2`) can define their own set (or list) of tricks. In fact, we expect that Maya can "roll over" and that Biff can "play dead".

However, take a look at the output:

```
I have a dog named Maya that can ['roll over', 'play dead'].
I have a dog named Biff that can ['roll over', 'play dead'].
```

The fact that both dog objects can do the same tricks can be explained by noting that the list `tricks` is defined at the class level and is therefore considered a class variable. As such, all instances of the dog class share the list. A change to it (even through the function `add_trick`) affects all instances of the dog class! To fix this and make the list of tricks an instance variable, we can define it in the `__init__` method as follows:

```
class Dog:
     kind = "canine"

     def __init__(self, dog_name):
          self.name = dog_name
          self.tricks = []

     def add_trick(self, trick):
          self.tricks.append(trick)

d1 = Dog("Maya")
d2 = Dog("Biff")
d1.add_trick("roll over")
d2.add_trick("play dead")

print "I have a dog named {} that can {}.".format(d1.name,\
       d1.tricks)
print "I have a dog named {} that can {}.".format(d2.name,\
       d2.tricks)
```

Since it is no longer at the class level, it is considered an instance variable and thus allows unique values to be stored for each instance of the dog class. Here is the output of the above modified Python code:

```
I have a dog named Maya that can ['roll over'].
I have a dog named Biff that can ['play dead'].
```

Now, take a look at this more complete dog class:

```
class Dog:
     kind = "canine"

     def __init__(self, name, breed):
          self.name = name
          self.breed = breed
          self.tricks = []
          self.friends = []

     def add_trick(self, trick):
          self.tricks.append(trick)

d1 = Dog("Maya", "mutt")
d2 = Dog("Biff", "Black Lab")
d1.add_trick("roll over")
d2.add_trick("play dead")

print "I have a dog named {} that can {}.".format(d1.name,\
       d1.tricks)
print "I have a dog named {} that can {}.".format(d2.name,\
       d2.tricks)

d1.friends = [ "Finca", "Shane" ]
d2.friends = [ "Sadie", "Bosco" ]
print "{}'s friends are {}.".format(d1.name, d1.friends)
print "{}'s friends are {}.".format(d2.name, d2.friends)
```

Note the addition of several new instance variables: `breed` and `friends`. This class defines all dogs to have a name, a breed, a list of tricks, and a list of friends.

Here is the program's output:

```
I have a dog named Maya that can ['roll over'].
I have a dog named Biff that can ['play dead'].
Maya's friends are ['Finca', 'Shane'].
Biff's friends are ['Sadie', 'Bosco'].
```

At this point, it may be worthwhile to summarize the difference between class variables, instance variables, and function parameters. Class variables are relevant to an entire class. The values of class variables are shared among all of the instances of a class. Think of a class variable as being stored in a single memory location that all the instances of a class can refer to. Instance variables are also relevant to an entire class. However, the values of instance variables are unique for each instance of a class. That is, an instance variable is stored in a different memory location for each instance of a class. Function parameters are relevant to a function and are only accessible inside the function. They are short-lived and last until the function has finished its execution.

Accessors and mutators

Consider the following simple dog class:

```
class Dog:
     kind = "canine"

     def __init__(self, name):
          self.name = name
          self.age = 0

d1 = Dog("Maya")
print "I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.age)
d1.age = -5
print "I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.age)
```

Now take a look at the output:

```
I have a dog named Maya that is 0 year(s) old.
I have a dog named Maya that is -5 year(s) old.
```

Everything seems to work fine; however, note that no dog can actually be -5 years old. This value is not possible for a dog's age (at least not in the world that we live in). This illustrates an important point: sometimes, we may want to check that the values supplied to function parameters are sensible. For numeric types, we typically call this **range checking**. That is, we may need to ensure that a supplied value falls within a valid range. For example, a valid range for a dog's age could be 0 to 29[2].

Range checking is a subset of a more general concept called **input validation**, which attempts to validate input (whether it be from a user during program execution, from actual parameters passed in to a function's formal parameters, etc). To ensure proper execution of a program that processes inputs, the inputs must first be validated. In the example above, the input to a dog's age must first be validated before the instance variable is assigned the value of the input.

To accomplish this, we can define a mutator (also known as a *setter*) that provides write access to an instance variable defined in a class. A **mutator** is a method that *wraps* an instance variable for the purpose of input validation (and often access control in some object-oriented programming languages). The instance variable still exists; however, to change it, the mutator must be called instead. Once the supplied input is validated, the instance variable is then changed with the provided value.

Here is a modified dog class with a mutator for the instance variable `age`:

```
class Dog:
     kind = "canine"

     def __init__(self, name):
          self.name = name
          self.age = 0

     def setAge(self, age):
          if (age >= 0 and age <= 29):
               self.age = age
```

[2] The oldest dog that ever lived was an Australian cattle dog named Bluey. He reached almost 29.5 years of age!

```
d1 = Dog("Maya")
print "I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.getAge())
d1.setAge(-5)
print "I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.getAge())
```

Note that the mutator is called `setAge`. Typically, we specifically set the name of a mutator to the word "set" followed by the name of the instance variable (initially capitalized). Since the mutator's purpose is to change the value of an instance variable, then that value must be passed in as a function parameter. The mutator then performs range checking. In the case of the dog class above, a value from 0 through 29 is valid (and would subsequently be assigned to the instance variable `age`). To change the value of a dog's age, the mutator must be called.

Here is the output now:

```
I have a dog named Maya that is 0 year(s) old.
I have a dog named Maya that is 0 year(s) old.
```

Note that the attempt to change `d1`'s age to -5 was not successful.

Using the function `setAge` as the mutator that enables modification of the instance variable `age` seems a bit tedious. In a perfect world, changing `d1`'s age (with input validation) would perhaps be done as follows:

```
d1.age = 11
```

However, doing it this way would effectively bypass the mutator, `setAge`, and ignore input validation (as seen in the earlier example). Python does provide a neat way to accomplish this, however. We often call this kind of *neat* behavior syntactic sugar. **Syntactic sugar** just means that a programming language provides a sensible (and often shorthand) way to accomplish a task that may, *under the hood*, be a bit more convoluted.

Python provides direct support for wrapping instance variables with mutators that perform input validation through a concept called a decorator. For now, a **decorator** is just a wrapper. It is something that wraps something else. In this case, it is a mutator in the form of a function that wraps an instance variable.

However, to properly explain how Python supports this, we must first discuss the concept of an accessor. An **accessor** (also known as a *getter*) is a method that wraps an instance variable for the purpose of providing *read* access (i.e., to allow us to read the the value of an instance variable). In Python, the meaning behind this is lost because all of a class' instance variables are directly accessible. However, in other object-oriented programming languages (such as Java, for example), we can enforce the privacy of instance variables. That is, we can restrict them such that they can only be accessed through accessors and mutators. Nevertheless, the only way that Python supports decorators as mutators is to additionally provide decorators as accessors. It may be best to first show the source code that demonstrates this:

```
class Dog(object):
     kind = "canine"
```

```
        def __init__(self, name):
             self.name = name
             self.age = 0

        # accessor
        @property
        def age(self):
             return self._age

        # mutator
        @age.setter
        def age(self, age):
             if (age >= 0 and age <= 29):
                  self._age = age

d1 = Dog("Maya")
print "I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.age)
d1.age = -5
print "I have a dog named {} that is {} year(s)\
       old.".format(d1.name, d1.age)
```

Note a few changes. First, the class header has changed from `class Dog:` to `class Dog(object):`. The actual meaning behind this will become clear later in this lesson when we discuss the concept of inheritance. Second, there are seemingly erroneous statements beginning with that "@" symbol (e.g., `@property` and `@age.setter`). In Python, these *tags* formally define decorators. The tag `@property` defines a decorator (or wrapper) that serves as an accessor, and the tag `@age.setter` defines a decorator (or wrapper) that serves as a mutator for a member called `age`.

Both the accessor and mutator are functions with the same name. In the case above, both are functions called `age`. Semantically, they refer to a dog's age. Since the identifier `age` is now used to refer to the accessor and mutator, the instance variable that these methods wrap must be renamed. In Python, it is typical to begin instance variables with an underscore. For example, the instance variable that stores a dog's age would be called `_age`.

Let's explain the accessor and mutator, one at a time. First, the accessor:

```
@property
def age(self):
     return self._age
```

Here, the tag `@property` defines a decorator that will serve as an accessor for the instance variable that represents a dog's age. The next statement defines the accessor itself. The function is called `age` (and only takes a single parameter, the object). Since the sole purpose of an accessor is to provide read access to an instance variable, then all that is required is to return its value (via the `return` keyword). Since the identifier `age` is used as the function's name, then the instance variable has been renamed to `_age` as noted earlier.

Now, the mutator:

```
@age.setter
def age(self, age):
    if (age >= 0 and age <= 29):
        self._age = age
```

Here, the tag `@age.setter` defines a decorator that will serve as a mutator for the instance variable that represents a dog's age. The next statement defines the mutator itself. The function is also called `age` (and takes two parameters: the object and the value to change the instance variable to). Since the purpose of a mutator is to provide write access to an instance variable with input validation, it appropriately ensures that the provided value is within an acceptable range. If so, the instance variable `_age` is changed to reflect the provided input value.

You may have noticed that the decorator for the mutator, `@age.setter`, contains the name of the function, `age`. This must be adhered to when defining a decorator as a mutator. If, for instance, we wished to provide a mutator for a dog's name, we could use the decorator tag `@name.setter`, call the mutator function `name`, and use the instance variable `_name`.

Note the following statement in the constructor:

```
self.age = 0
```

Be careful! Here, `self.age` does not refer to an instance variable. It actually refers to the mutator. This assignment statement effectively calls the mutator, passing in the value on the right-hand side (0) as the second parameter of the mutator (`age`). That is, the value 0 is passed in to the mutator, which is then validated in the mutator. Since it is within the acceptable range (0 through 29), then the instance variable `_age` is set to 0.

It is important to note that the accessor must be defined **before** the mutator. Using the `@property` tag defines the property by name (e.g., `age`) so that it can be used to subsequently define the mutator (`@age.setter`).

To illustrate accessors and mutators a bit more, consider the following class that defines a 2D point (with an x- and y-coordinate):

```
# points must fall within the range (-10,-10) and (10,10)
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    # getter for x
    @property
    def x(self):
        return self._x

    # setter for x
    @x.setter
    def x(self, value):
```

```
            if (value < -10):
                self._x = -10
            elif (value > 10):
                self._x = 10
            else:
                self._x = value

        # getter for y
        @property
        def y(self):
            return self._y

        # setter for y
        @y.setter
        def y(self, value):
            if (value < -10):
                self._y = -10
            elif (value > 10):
                self._y = 10
            else:
                self._y = value

p1 = Point()
p2 = Point(5, 5)
p3 = Point(-50, 50)

print "p1=({},{})".format(p1.x, p1.y)
print "p2=({},{})".format(p2.x, p2.y)
print "p3=({},{})".format(p3.x, p3.y)
```

Although the class for a 2D point is a bit more involved, it only contains two instance variables. The first, `_x`, represents the position of the point in the x-direction. The second, `_y`, represents the position of the point in the y-direction. Accessors and mutators for each are provided (via the methods called `x` for the instance variable `_x`, and the methods called `y` for the instance variable `_y`). In addition, range checking is performed for both the x- and y-components. Each component may not be less than -10 or greater than 10. Here is the output of the program:

```
p1=(0,0)
p2=(5,5)
p3=(-10,10)
```

Notice that the input validation works (i.e., declaring the point `p3` at -50,50 results in a point initialized at -10,10).

Did you notice something odd in the constructor? Here it is for reference:

```
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Take a look at the constructor's parameters: `(self, x=0, y=0)`. You probably expected something more like this: `(self, x, y)`. In Python, we can provide **default** values for function parameters. This works for any function (not just the constructor). This means that, should parameter values be unspecified when the function is called, the default values will be used. In this example, a point's default x- and y-values are 0 and 0 respectively. Therefore, a point at the origin could be instantiated as follows:

```
p = Point()
```

Of course, such a point could also be instantiated as follows:

```
p = Point(0, 0)
```

The default values are only used if parameter values are not specified when the function is called. It is important to note that a function can have a mix of both standard and default parameters. In fact, the constructor is just like this (`self` is a standard parameter without a default value, while `x` and `y` have default values). In Python, all parameters with default values must be specified after standard parameters. This way, it is clear if values for default parameters are specified in a function call. For example:

```
def foo(a, b, c, d=5, e=7, f=8):
     pass

...

foo(1, 2, 3, 4)
```

In this case, the actual parameters 1, 2, 3, and 4 are mapped to the formal parameters `a`, `b`, `c`, and `d`. The default values for the formal parameters `e` and `f` are used.

To wrap up this section, let's add line numbers to the point class above and trace the program's execution:

```
 1:  # points must fall within the range (-10,-10) and (10,10)
 2:  class Point(object):
 3:       def __init__(self, x=0, y=0):
 4:            self.x = x
 5:            self.y = y

 6:       # getter for x
 7:       @property
 8:       def x(self):
 9:            return self._x

10:       # setter for x
11:       @x.setter
12:       def x(self, value):
13:            if (value < -10):
14:                 self._x = -10
15:            elif (value > 10):
16:                 self._x = 10
17:            else:
```

```
18:                 self._x = value

19:         # getter for y
20:         @property
21:         def y(self):
22:             return self._y

23:         # setter for y
24:         @y.setter
25:         def y(self, value):
26:             if (value < -10):
27:                 self._y = -10
28:             elif (value > 10):
29:                 self._y = 10
30:             else:
31:                 self._y = value

32:  p1 = Point()
33:  p2 = Point(5, 5)
34:  p3 = Point(-50, 50)

35:  print "p1=({},{})".format(p1.x, p1.y)
36:  print "p2=({},{})".format(p2.x, p2.y)
37:  print "p3=({},{})".format(p3.x, p3.y)
```

In the space below, trace the execution path of the program by listing the lines numbers:

Did you know?

Although some object-oriented languages actually prevent accessing instance variables that are protected (or wrapped) with accessors and mutators, Python does not enforce this. For example, it is possible to change the x-component or access the y-component of a point via statements such as:

```
p1._x = -22
print p1._y
```

Many Python programmers prefer to change the way they implement classes so that any value that requires protection is not stored in instance variables. There are other mechanisms that will, in fact, protect these values. However, this discussion is beyond the scope of this lesson.

Activity 2: Fractions

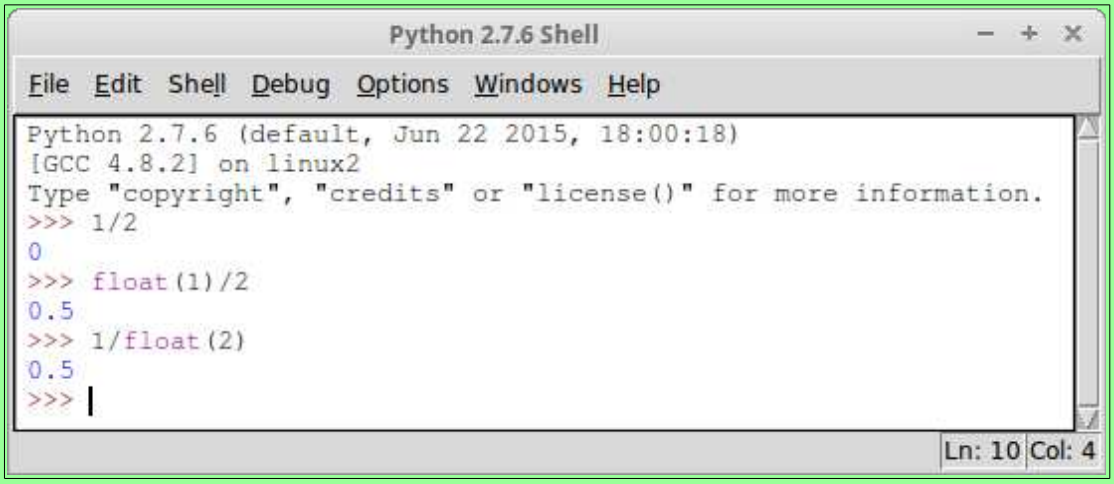
In this activity, we will create a class that represents a fraction. The first step is to determine what makes up a fraction (i.e., what it can *be* – its state). This task is pretty simple! Fractions have a numerator and a denominator. We can create instance variables for these and also provide accessors and mutators for each.

We may also want to provide the numeric representation of a fraction. For example, the fraction 1/2 has the numeric representation 0.5. In Python, simply dividing the numerator by the denominator (i.e., 1/2) won't produce the anticipated result because it will perform integer division. That is, the expression 1/2 will result in 0 (since 2 goes into 1 exactly 0 times). To produce a floating point result, we must convert one of the two operands to a floating point value. In Python, we can do this as follows:

```
float(1) / 2
```

The expression `float(1)` converts the integer value 1 into the floating point value 1.0. Generally, the expression `float(x)` converts the operand *x* into a floating point value. Formally, this conversion is called a **typecast**, in that the operand's *type* is *cast* to a different type.

The expression `float(1) / 2` produces the expected result (0.5). In fact, typecasting either operand works – as shown in the example below:



There are other typecast operators that perform various type conversions. Here are a few of them:

- `int(x)` – converts *x* to an integer
- `long(x)` – converts *x* to a long integer
- `complex(x, y)` – creates a complex number; *x* is the real part, *y* is the imaginary part
- `str(x)` – converts *x* to a string

Lastly, we must not allow the denominator of a fraction to ever be 0 (since division by 0 is mathematically illegal). Therefore, we will need to provide range checking (via if-statements, for example), to ensure that such an assignment is prevented.

Here's the beginning of the fraction class, along with a brief main program to test the class:

```
# defines a fraction
class Fraction(object):
     # by default, a fraction is 0/1
     def __init__(self, num = 0, den = 1):
          self.num = num
          # make sure not to set the denominator to 0 if
          #  specified
          if (den == 0):
               den = 1
          self.den = den

     # getter for the numerator
     @property
     def num(self):
          return self._num

     # setter for the numerator
     @num.setter
     def num(self, value):
          self._num = value

     # getter for the denominator
     @property
     def den(self):
          return self._den

     # setter for the denominator
     @den.setter
     def den(self, value):
          # ignore if the specified denominator is 0
          if (value != 0):
               self._den = value

     # returns a fraction's numeric representation
     def getReal(self):
          return float(self.num) / self.den

# main program
f1 = Fraction()
f2 = Fraction(1, 2)
f3 = Fraction(0, 0)

print "{}/{} ({})".format(f1.num, f1.den, f1.getReal())
print "{}/{} ({})".format(f2.num, f2.den, f2.getReal())
print "{}/{} ({})".format(f3.num, f3.den, f3.getReal())
```

And here's the output:

```
0/1 (0.0)
1/2 (0.5)
0/1 (0.0)
```

Note how the class prevents the third fraction from being initialized as 0/0 and, instead, changes it to 0/1.

Did you know?

There is a better way of displaying a fraction than what is shown in the example above. Note how we earlier structured a print statement that built the string representation of a fraction:

```
print "{}/{} ({})".format(f1.num, f1.den, f1.getReal())
```

In Python, we can define a built-in **magic function** that is automatically called when we wish to display an object. In fact, this built-in function is user-definable and is named using a similar format as the constructor (i.e., the function begins and ends with two underscores). The function is called `__str__` and must return a string representation of the class. So for a fraction, such a function could be implemented as follows:

```
def __str__(self):
    return "{}/{} ({})".format(self.num, self.den, self.getReal())
```

Displaying a fraction would then be possible via the following much simpler statement (via syntactic sugar):

```
print f1
```

Adding this function to the fraction class is simple. Here's a snippet of the addition:

```
# defines a fraction
class Fraction(object):

    ...

    # returns a fraction's string representation
    def __str__(self):
        return "{}/{} ({})".format(self.num, self.den,\
               self.getReal())


...

# main program
f1 = Fraction(1, 2)
f2 = Fraction(1, 4)
f3 = f1.add(f2)

print f1
print f2
print f3
```

Of course, the output is the same as before!

Activity 3: Reducing fractions

You may have noticed that instantiating the fraction 6/8 would work just fine. The problem is that this fraction is not expressed in lowest terms. That is, it can be reduced (to 3/4). Our fraction class would greatly benefit from a function that can reduce a fraction. Such a function could be called in the constructor after setting the numerator and denominator in case it is not in lowest terms.

Although there are many ways to reduce a fraction, here's a simple algorithm that calculates the greatest common divisor (GCD) among the numerator and denominator. First, initially assume that the GCD is 1. From there, iterate, starting with 2 through the smaller of the numerator or denominator. Each time, the objective is to try to find a value that evenly divides both the numerator and denominator. As such a value is found, the GCD is updated. The final step is to divide the numerator and denominator by the GCD (which reduces the fraction). As a cleanup operation, if the numerator is 0 (i.e., the fraction's numeric value is 0.0), the denominator is set to 1 (i.e., 0/1).

This is shown in the snippet of code below (which can be placed anywhere in the fraction class):

```
    # reduces a fraction
    def reduce(self):
        # we initially assume that the GCD is 1
        # from there, we iterate starting at 2 through the smaller
        #  of the numerator or denominator
        # since the numerator and denominator could be negative,
        #  we use their absolute values
        # each time, we try to find a value that evenly divides
        #  both the numerator and denominator
        # as we find such a value, we update the GCD
        # the final step is two divide the numerator and
        #  denominator by the GCD to reduce the fraction
        # as cleanup, if the numerator is 0 (i.e., the fraction is
        #  0) then set the denominator to 1
        gcd = 1
        minimum = min(abs(self.num), abs(self.den))

        # find common divisors
        for i in range(2, minimum + 1):
            # when we find one, update the GCD
            if (self.num % i == 0 and self.den % i == 0):
                gcd = i

        # divide the numerator and denominator by the GCD
        self.num /= gcd
        self.den /= gcd

        # if the numerator is 0, set the denominator to 1
        if (self.num == 0):
            self.den = 1
```

The Python *math* library has many useful functions. The **min** function returns the minimum value of a number of values passed in as parameters. This makes it quite easy to determine which of the numerator or denominator is smaller. Since a function's numerator or denominator could be negative, we use their absolute value to determine which is smaller. The **abs** function returns the absolute value of a specified value.

So where (and when) do we call the `reduce` function? For the fraction class shown earlier, we could do so in the constructor as follows:

```
# defines a fraction
class Fraction(object):
    # by default, a fraction is 0/1
    def __init__(self, num = 0, den = 1):
        self.num = num
        # make sure not to set the denominator to 0 if\
        #  specified
        if (den == 0):
            den = 1
        self.den = den

        self.reduce()
```

This works; however, the mutators for the numerator and denominator may cause a fraction to no longer be reduced. We could, therefore, call the `reduce` function at the end of the mutators. There is a problem with this, however. Since the constructor uses the mutator for the numerator (in `self.num = num`) before the denominator is even set, a call to the function `reduce` in the mutator for the numerator would attempt to access the denominator (which doesn't exist). This would result in an error.

We could try to place a call to the `reduce` function in the mutator for the denominator. But this is also problematic, because the `reduce` function uses the mutator for the denominator to change the denominator (i.e., when dividing it by the *gcd*). Placing it here would cause the `reduce` function to be recursively called infinitely!

Perhaps the best place to put the call to the `reduce` function is in the `__str__` function (i.e., when displaying a fraction).

Activity 4: Adding fractions...and more

Let's now implement the functionality to add two fractions and produce the sum of these as a new fraction. We must first discuss how two fractions can be added. Typically, the least common denominator is found. A simpler version, however, is to multiply each fraction by the other's denominator to obtain a common denominator (that is not necessarily the least common denominator). Here's an illustration:

$$\frac{a}{b}+\frac{c}{d} = \frac{a*d}{b*d}+\frac{b*c}{b*d}$$

As an example, take the following:

$$\frac{1}{2}+\frac{1}{4} = \frac{1*4}{2*4}+\frac{2*1}{2*4} = \frac{4}{8}+\frac{2}{8} = \frac{6}{8} = \frac{3}{4}$$

So now, how do we implement a method in the fraction class that does this? One way is as follows:

```
# calculates and returns the sum of two fractions
def add(self, other):
    num = (self.num * other.den) + (other.num * self.den)
    den = self.den * other.den
    sum = Fraction(num, den)
    sum.reduce()

    return sum
```

This function is called as follows (specifically in the third statement below):

```
f1 = Fraction(1, 2)
f2 = Fraction(1, 4)
f3 = f1.add(f2)
```

Note that both fractions are effectively passed in to the `add` function. The first, `f1`, represents the current instance and is mapped to the first parameter, `self`. The second, `f2`, is mapped to the second parameter, `other`.

The function implements the common denominator method shown above and generates a fraction representing the sum of `self` and `other`. The new fraction is then reduced and returned. Note that calling the `reduce` function here is not necessary if it is called in the `__str__` function.

The function could be optimized (perhaps at the expense of not being quite as readable) by returning a new fraction directly instead of creating one and returning it. For example:

```
return Fraction(num, den)
```

In fact, the function could be optimized even more as follows:

```
return Fraction(self.num * other.den + other.num * self.den,\
    self.den * other.den)
```

Of course, we could implement functions to subtract, multiply, and divide fractions! In fact, we could implement a subtract method by using the already defined add method. How? Recall that subtracting is just adding the negative. Since this will be assigned as a program later, it is left as an individual exercise for now.

Operator overloading

In the last example above, we defined a function in the fraction class that adds two fractions and returns the sum. We used this function similar to the following snippet of Python code:

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)
print f1.add(f2)
```

As expected, the output of these statements is `5/4 (1.25)`.

The way in which we call the addition function seems a bit tedious. Why can't we just use the addition operator? For example, why can't we just add two fractions, `f1` and `f2`, by merely using the expression `f1 + f2`? This would mean that the following modification of the example above would work as expected:

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)
print f1 + f2
```

It turns out that such a thing is possible through a concept called operator overloading. **Operator overloading** is the act of redefining the behavior of operators (such as addition and subtraction) using their known symbols (+ for addition, – for subtraction, and so on) in order to support these operations on user-defined data types. For example, redefining the addition operator for the fraction class could mean implementing the common denominator method described earlier.

Python has various internal magic functions that support the redefinition of common operators. The main idea is to encapsulate the new, redefined behavior in a function that is automatically called (using syntactic sugar) when two objects of the class are used as operands with the specified operator. For the purpose of the fraction class, we will only consider the four arithmetic operators.

The addition operator (+) can be redefined in a function called `__add__` as follows:

```
def __add__(self, other):
    num = (self.num * other.den) + (other.num * self.den)
    den = self.den * other.den
    sum = Fraction(num, den)
    sum.reduce()

    return sum
```

Note that the source code in the new overloaded `__add__` function is exactly the same as it was in the original `add` function shown earlier.

The subtraction operator (–) can be redefined in a function called `__sub__` as follows:

```
def __sub__(self, other):
    ...
```

Note that for this and the remaining operators, the source code is not provided. Instead, appropriate code is replaced with an ellipsis (…).

The multiplication operator (*) can be redefined in a function called `__mul__` as follows:

```
def __mul__(self, other):
    ...
```

Lastly, The division operator (/) can be redefined in a function called `__div__` as follows:

```
def __div__(self, other):
    ...
```

The fraction class has now grown! Take a look:

```
# defines a fraction
class Fraction(object):
     # by default, a fraction is 0/1
     def __init__(self, num = 0, den = 1):
          self.num = num
          # make sure not to set the denominator to 0 if\
          #  specified
          if (den == 0):
               den = 1
          self.den = den

          self.reduce()

     ...

     # calculates and returns the sum of two fractions
     def __add__(self, other):
          num = (self.num * other.den) + (other.num * self.den)
          den = self.den * other.den
          sum = Fraction(num, den)
          sum.reduce()

          return sum

     # calculates and returns the difference of two fractions
     def __sub__(self, other):
          # replace this with the function's actual\
          #  implementation
          return None

     # calculates and returns the product of two fractions
     def __mul__(self, other):
          # replace this with the function's actual\
          #  implementation
          return None

     # calculates and returns the division of two fractions
     def __div__(self, other):
          # replace this with the function's actual\
          #  implementation
          return None

     ...
```

In fact, we can now perform all of the implemented arithmetic operations on fractions. Here's a snippet of Python code that tests the fraction class and assumes that the operator overload functions have been fully implemented:

```
     # main program
```

```
f1 = Fraction(1, 2)
f2 = Fraction(1, 4)

print f1
print f2
print f1 + f2
print f1 - f2
print f1 * f2
print f1 / f2
```

And here is the output:

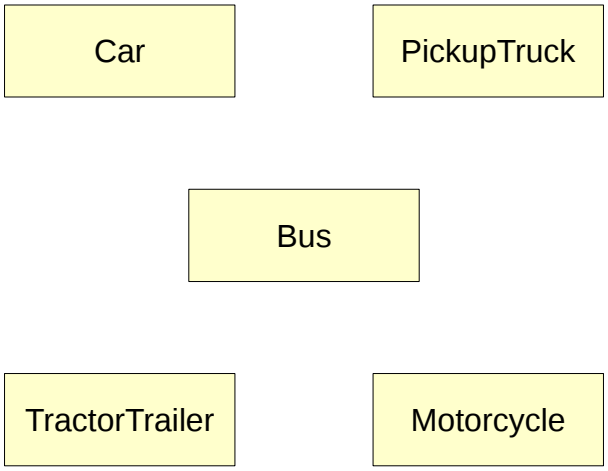
```
1/2 (0.5)
1/4 (0.25)
3/4 (0.75)
1/4 (0.25)
1/8 (0.125)
2/1 (2.0)
```

Class diagrams

In computer science courses, you are often asked to design simple programs. The ability to understand and hold everything in one's head when solving a simple problem is relatively straightforward and, well, usually pretty simple. However, when solving complicated problems and developing solutions to these problems as large and tedious applications, it becomes quite difficult to manage all of the parts and pieces. Often, we require the use of tools and techniques that incorporate visual aids and diagrams to assist us in managing the structure and components of these applications.

A **class diagram** is a type of diagram that describes the structure of a program by visualizing its classes, their state and behavior, and their relationships. The most simple class diagram only shows the classes of a program, which are represented as rectangles.

To illustrate how a class diagram could be used to model an application's structure, let's consider one that models vehicle traffic in a large city for the purpose of analyzing how it manages traffic during rush hour. This kind of application would be useful in learning about traffic patterns, congestion, and so on. In fact, it could help to redesign roads, entrances to and exits from highways and interstates, the placement and timing of traffic signals, etc. Such an application may include classes for cars, pickup trucks, buses, tractor trailers, motorcycles, and so on, since all of these things contribute to the traffic in a city. In fact, the application could be modeled with a class diagram as follows:



The classes of an application are always singular nouns. Since a class is a blueprint for objects, then a class is essentially like a rubber stamp. For example, we can define a class that describes the blueprint for a car. This class would be considered the car class and be formally called **Car**. As mentioned earlier, the names of classes are typically capitalized. Since they are identifiers, they also must not contain spaces and abide by all of the rules for naming identifiers in the programming language. In Python, the car class could be defined as follows:

```
class Car:
    ...
```

Instances of the car class would collectively be called cars (and there could be many of them). Similarly, the class for a pickup truck could be called **PickupTruck**, and would be defined in Python as follows:

```
class PickupTruck:
    ...
```

The beauty of a class diagram is that it allows us to very easily see the components of a system or application. In the class diagram above, there is no indication of the state and behavior of classes, nor is there any indication of any relationships between classes. We will get to this later.

Inheritance

As you know, the object-oriented paradigm attempts to mimic the real world, particularly in how it is made up of objects that interact with each other. In the real world, objects also have relationships, and this is useful! For example, a person inherits traits from parents. Specifically, a person inherits physical traits (e.g., height, hair color, etc) and behavioral traits (e.g., manner of speaking). This behavior is represented in the object-oriented paradigm as well.

To illustrate how this is done, let's consider the application that models vehicle traffic described earlier. Let's begin with the car class that serves as the blueprint for a car in the traffic simulation. What might its state and behavior look like? That is, what are cars made up of? What can they be, and what can they do? Very quickly, we can think of attributes such as year, make, model, mileage, and so on. This represents the state of a car. We can also think of behaviors such as start, move, turn, park, and so on. In fact, we could quickly design a car class now that we know how to do so in Python!

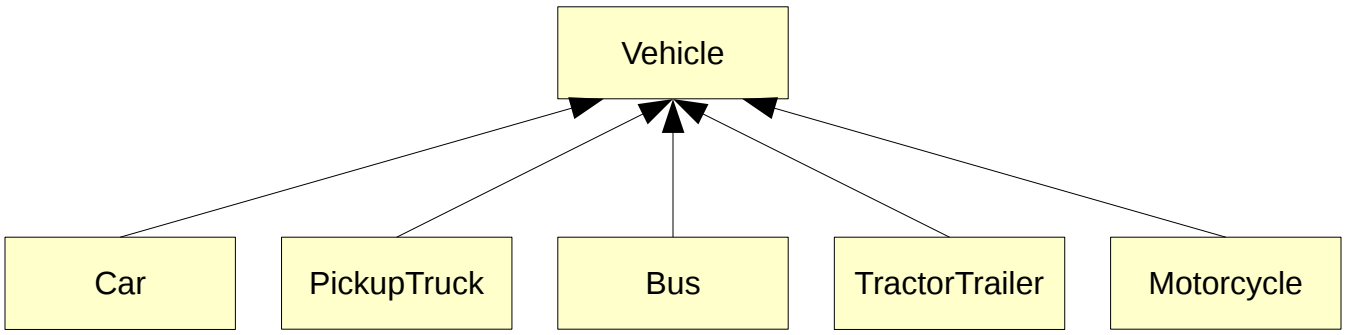
Now let's consider a pickup truck class that serves as the blueprint for a pickup truck in the traffic simulation. Its state would most likely be very similar to that of the car class. And so would its

behavior. In fact, not much differentiates a car from a pickup truck. They both generally have the same attributes and do the same thing. Imagine designing the classes for a car and a pickup truck. You may think that the classes would share many similarities in both state and behavior (and you would be right).

Now imagine maintaining such an application. Suppose that the implementation of some behavior that is similar across cars and pickup trucks needs to be modified. This would require changing both the car and pickup truck classes because code is duplicated across the two classes. Dealing with this type of thing increases the likelihood of bugs. The beauty of the object-oriented paradigm is that it allows the inheritance of state and behavior from class-to-class, just like we inherit traits from our ancestors!

The state and behavior that is shared among the car and pickup truck classes in the traffic simulation application could be captured in a more general class. Such a class could, for example, be called a **Vehicle**. All of the state and behavior that is shared among any type of vehicle would be defined in this class. Specific kinds of vehicles (like cars and pickup trucks) would then inherit these *traits*. Any modifications to the state and behavior of vehicles of all types could be made in the vehicle class and be automatically applied to all types of vehicles!

In fact, let's amend the class diagram shown earlier by including a vehicle class that defines the overall state and behavior that all types of vehicles (cars, pickup trucks, buses, tractor trailers, and motorcycles) share:



Note how all of the classes that inherit state and behavior from the vehicle class now have solid arrows pointing *toward* the vehicle class. In a class diagram, this indicates an inheritance relationship. Specifically, the car, pickup truck, and other classes shown at the bottom of the class diagram inherit state and behavior from the vehicle class. A class that defines state and behavior that is inherited by other classes is called a **superclass**. The classes that inherit from it are called **subclasses**. In the class diagram above, the class **Vehicle** is a superclass of the class **Car**, and the class **Car** is a subclass of the class **Vehicle**.

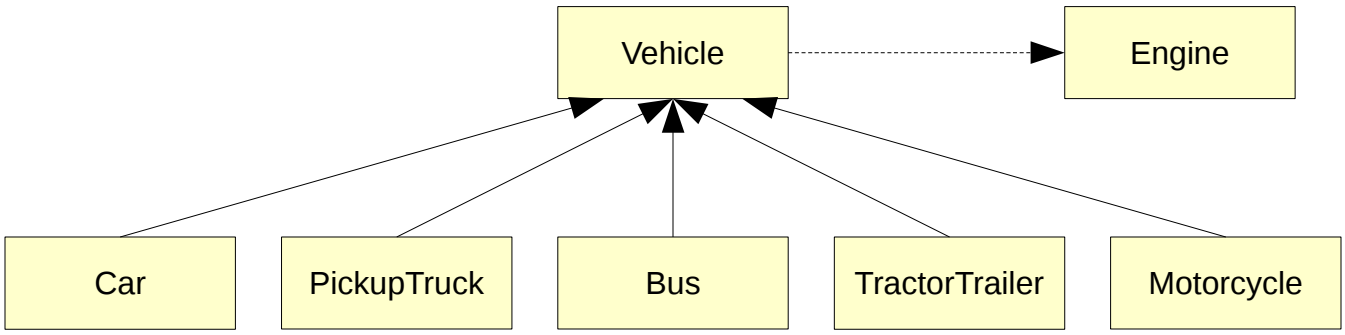
The inheritance relationship is often called the **is-a** relationship. This is actually quite clear from the class diagram: a Car *is a* Vehicle, a Bus *is a* Vehicle, and so on. There is also the **has-a** relationship. This represents a composition relationship and refers to the state of an object. Specifically, we often note the has-a relationship in class diagrams for classes that contain other classes.

In terms of how this is accomplished in Python source code, we merely need to specify the superclass in a subclass' class definition. For example, consider the class **Car** (which is a subclass of the superclass **Vehicle**). To note this relationship in Python, we merely need to define the **Car** class as follows:

```
class Car(Vehicle):
    ...
```

This establishes the relationship that the class **Car** is a subclass of the class **Vehicle**, and that the class **Vehicle** is a superclass of the class **Car**.

Next, consider an engine class that defines everything that an engine can be and do. Clearly, a car has an engine. So does a pickup truck, a bus, a motorcycle, and so on. In general, all of these vehicles have an engine. Since all vehicles have an engine, in the design of the application we may include the engine class as part of the state of the vehicle class. Specifically, we would include an instance of the engine class in the vehicle class. All subclasses of the vehicle class would then inherit this attribute. We note the has-a relationship in a class diagram with a dashed arrow that point toward the composed class. Here is an amended class diagram that now includes the engine class:

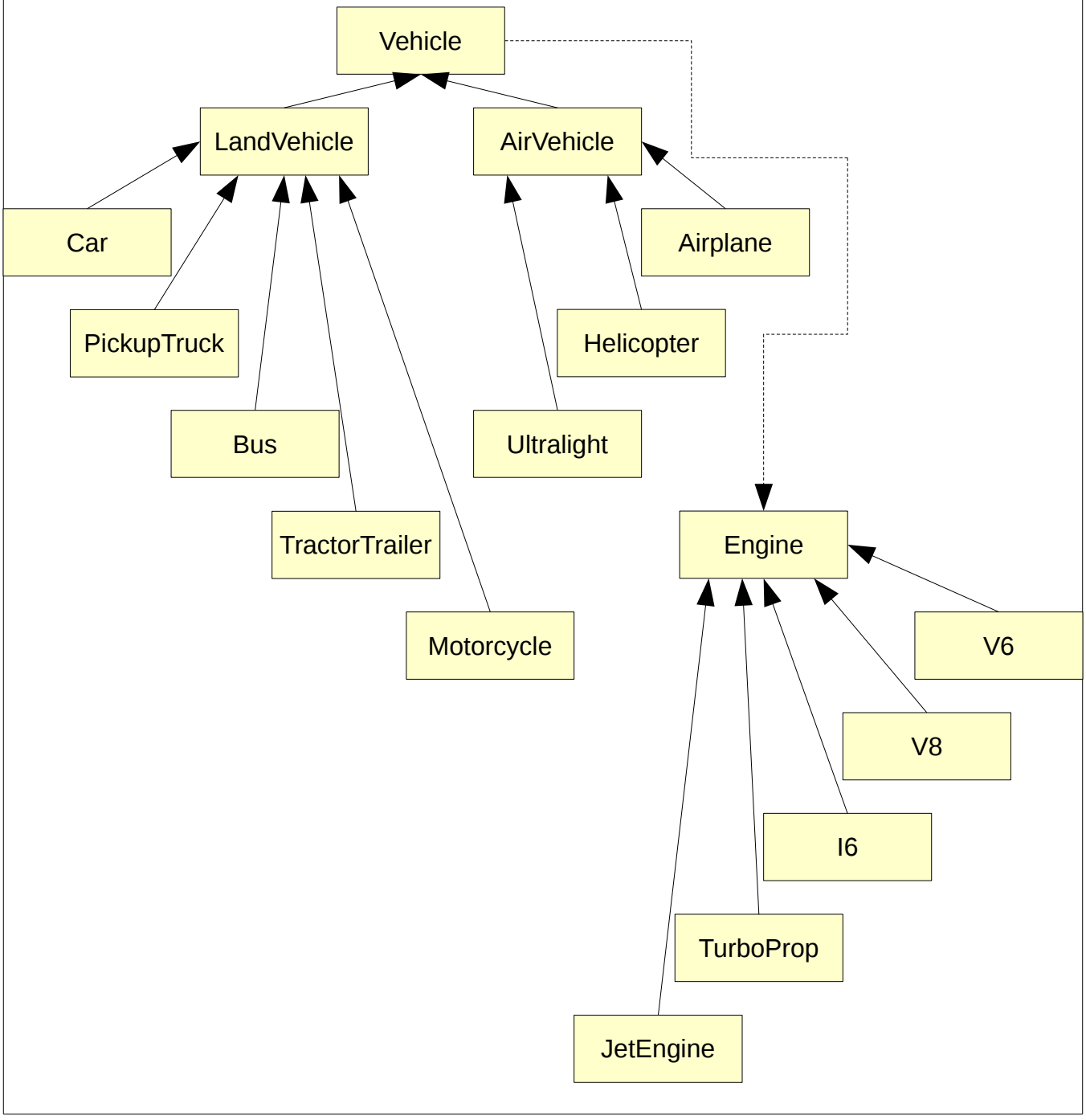


This important relationship illustrates that objects can, in fact, create other objects! In the example above, a vehicle can create an instance of an engine. Although the state and behavior of all vehicles is defined in the vehicle class, nothing stops any of its subclasses from redefining or specializing these attributes or behaviors. That is, although a car and a motorcycle both have an engine, they are quite different. Simply because the engine class is included in the vehicle class does not prevent a car or a motorcycle from specializing the engine and uniquely setting its state.

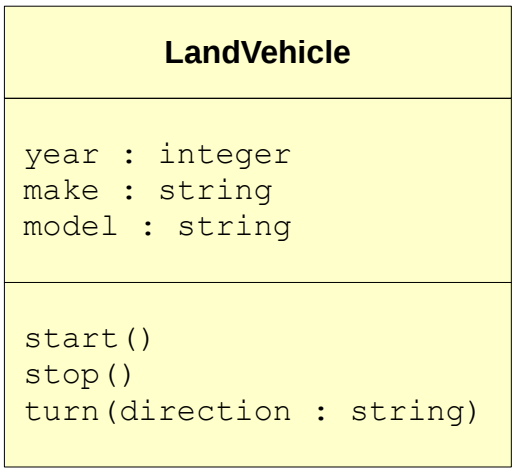
Let's further illustrate the concept of inheritance by expanding the world of vehicles. In this expanded world, there are two types of vehicles: land vehicles (that move on land) and air vehicles (that fly in the air). The types of land vehicles that exist include all of the vehicles described earlier (e.g., cars, pickup trucks, etc), and the types of air vehicles that exist include airplanes, helicopters, and ultralights. While we're at it, let's define multiple types of engines for land vehicles (e.g., V-6, V-8, and I-6), and multiple types of engines for air vehicles (e.g., turbo prop and jet engine).

Try to represent this expanded world with a class diagram in the space below:

Here is one possible class diagram:



Often, we include the state and behavior of classes in the class diagram. Suppose that the class **LandVehicle** has the instance variables `year`, `make`, and `model`, and the functions `start`, `stop`, and `turn`. The class diagram for this single class would be extended as follows:



Typically, we include the types of instance variables and adhere to the following format:

```
variable_name : variable_type
```

For functions, we include the names and types of any parameters and adhere to the following format:

```
function_name(parameter1_name : parameter1_type, …)
```

You have probably noticed that this extension of class diagrams makes it quite easy to implement the source code for the class!

The `object` class

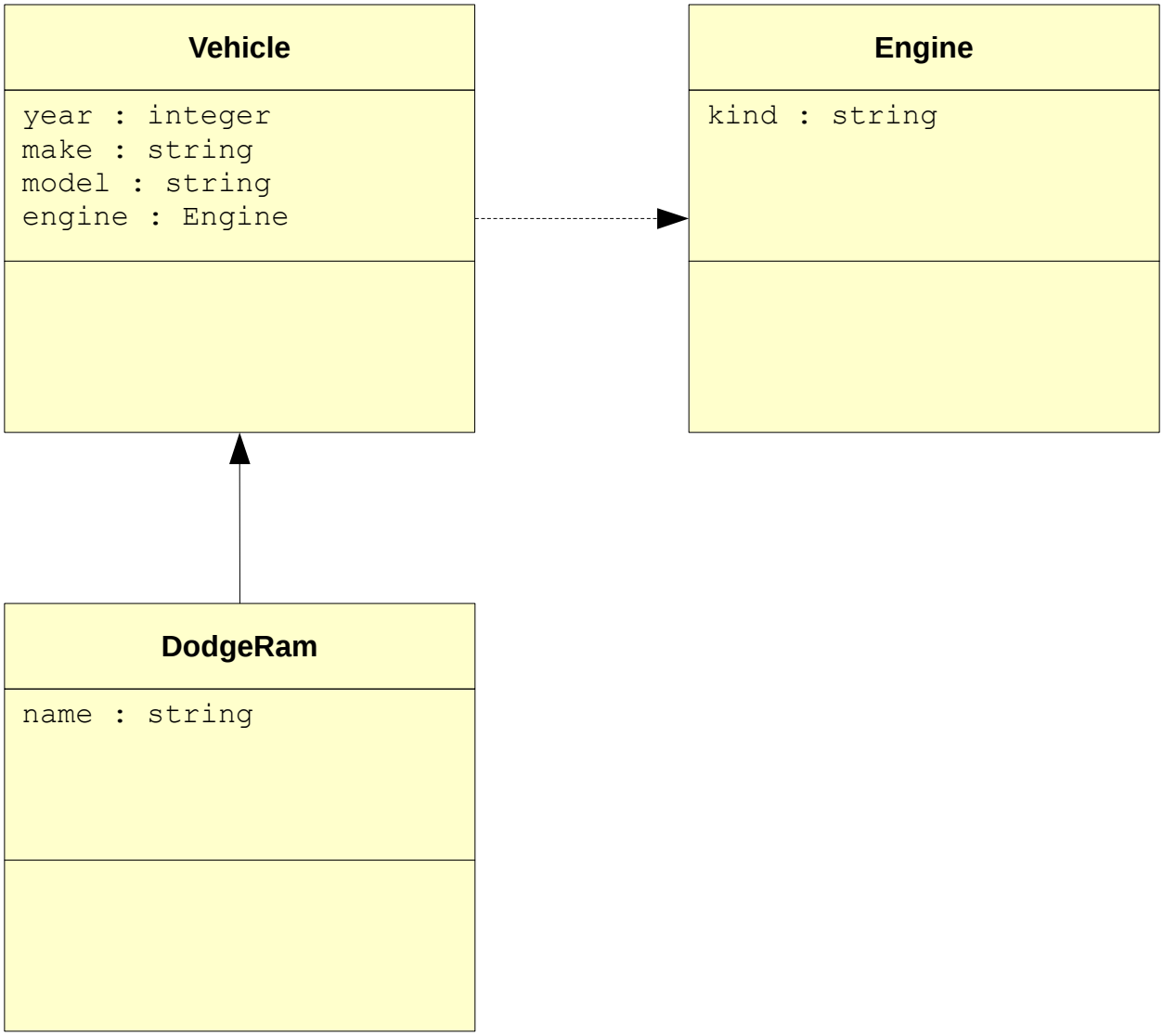
The inheritance relationship is easily implemented in Python classes in the class header. Earlier, when discussing accessors and mutators (with the dog class), we noted that the class definition must include what you now know to be an inheritance relationship with a class called **object**. In fact, here was the class header for the dog class:

```
class Dog(object):
    ...
```

Formally, the class **object** is defined to be the ultimate superclass for all built-in (i.e., user-defined) types. As shown, it is possible to have multiple levels of inheritance (e.g., the class **Car** is a subclass of the class **LandVehicle** which is a subclass of the class **Vehicle**). At the top of this inheritance hierarchy lies the object class. Although it is not strictly necessary, the class header for the vehicle class can be:

```
class Vehicle(object):
    ...
```

To support the syntactic sugar method of implementing accessors and mutators, an inheritance relationship must be specified. For the dog class, we needed to specify the **object** class as its superclass. In the case of the class **Car**, for example, it already inherits from the class **LandVehicle**. This is sufficient to support accessors and mutators using syntactic sugar. To better illustrate this, let's consider the following class diagram:



From this class diagram, we can quickly begin laying out the source code for the classes. In fact, the class headers can be directly inferred from the class diagram:

```
class Vehicle(object):
      …

class DodgeRam(Vehicle):
      …

class Engine(object):
      ...
```

Since the class diagram includes the state of each class, declaring instance variables in the constructors of each class and providing appropriate accessors and mutators is also relatively straightforward. In fact, here's the entire **Vehicle** class:

```
class Vehicle(object):
     def __init__(self, year, make, model):
          self.year = year
          self.make = make
          self.model = model
          self.engine = None

     @property
     def year(self):
          return self._year

     @year.setter
     def year(self, value):
          self._year = None
          if (value > 1800 and value < 2018):
               self._year = value

     @property
     def make(self):
          return self._make

     @make.setter
     def make(self, value):
          self._make = value

     @property
     def model(self):
          return self._model

     @model.setter
     def model(self, value):
          self._model = value

     @property
     def engine(self):
          return self._engine

     @engine.setter
     def engine(self, value):
          self._engine = value

     def __str__(self):
          return "Year: {}\nMake: {}\nModel: {}\nEngine:\
                  {}".format(self.year, self.make, self.model,\
                  self.engine)
```

The constructor includes parameters for a vehicle's year, make, and model. By default, a vehicle's engine is undefined (i.e., `None`). The class contains getters and setters for each of the instance variables. Finally, the `__str__` function defines how to represent a vehicle as a string, which is in the following format (with an example for clarity):

```
Year: 2016
Make: Dodge
Model: Ram
Engine: V6
```

Here is the entire **Engine** class:

```
class Engine(object):
    def __init__(self, kind=None):
        self.kind = kind

    @property
    def kind(self):
        return self._kind

    @kind.setter
    def kind(self, value):
        self._kind = value

    def __str__(self):
        return str(self.kind)
```

The **Engine** class has a single instance variable (its kind). Its constructor includes one parameter for the kind of engine (`None` by default). A getter and setter is provided for the instance variable. The string representation of an engine is just its kind.

Lastly, here is the **DodgeRam** class:

```
class DodgeRam(Vehicle):
    make = "Dodge"
    model = "Ram"

    def __init__(self, name=None, year=None):
        Vehicle.__init__(self, year, DodgeRam.make,\
                         DodgeRam.model)
        self.name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value
```

```
    def __str__(self):
        return "Name: {}\n{}".format(self.name,\
                Vehicle.__str__(self))
```

The **DodgeRam** class has a single instance variable (its name). Its constructor takes two parameters (for a DodgeRam's name and year). Note that the class contains two class variables that are shared among all instances of the class: make and model. This makes sense, because all instances of the class **DodgeRam** are Dodge Rams! That is, their make is Dodge, and their model is Ram.

There are two more interesting (and new) things in the class **DodgeRam**. Take a look at the first statement in the constructor:

```
Vehicle.__init__(self, year, DodgeRam.make, DodgeRam.model)
```

When implementing inheritance relationships, it often becomes useful and sometimes necessary to invoke or call functions in a subclass' superclass. Formally, state and behavior that are defined in the superclass are inherited in a subclass. They can be redefined in the subclass; however, they don't necessarily need to be. In fact, a subclass may inherit a function and need to implement the inherited behavior first. This is accomplished by calling the function in the superclass. Since the current object (self) is not an instance of the superclass, then invoking a function in the superclass is done by using the superclass' name. So the first part of the statement, `Vehicle.__init__`, means to call the constructor in the **Vehicle** class (the superclass of the **DodgeRam** class).

In this case, the year, make, and model are passed as parameters to the constructor in the **Vehicle** class. This effectively sets up the appropriate instance variables in the **Vehicle** class (which are inherited in the **DodgeRam** class). Subsequently, the constructor in the **DodgeRam** class then initializes its only instance variable, `name`.

Did you know?

There is another way of referring to a superclass by using a built-in function called `super`. This function takes two parameters: the name of the subclass and the instance of the subclass, `self`. For example, in the subclass **DodgeRam**, referring to the superclass **Vehicle** could be accomplished as follows:

```
super(DodgeRam, self)
```

To then invoke a function in the superclass, we simply append the function name and any parameters (other than `self`) to the `super` function; for example:

```
super(DodgeRam, self).__init__(year, DodgeRam.make, DodgeRam.model)
```

Note that this applies to Python version 2.7.x. An updated syntax is provided in Python version 3 that is beyond the scope of this lesson.

Another new thing in the class is the statement in the `__str__` function:

```
return "Name: {}\n{}".format(self.name, Vehicle.__str__(self))
```

The string representation of a **DodgeRam** is its name, followed by the string representation of a **Vehicle** (which was illustrated earlier). The latter part of the statement calls the `__str__` function in the Vehicle class:

```
        Vehicle.__str__(self)
```

Again, this illustrates a call to a function in the superclass. This call returns the string representation of a vehicle – which is displayed below the name of the **DodgeRam**; for example:

```
Name: Boss Hog
Year: 2016
Make: Dodge
Model: Ram
Engine: V6
```

So the string representation of a **DodgeRam** is simply its name, followed by the inherited string representation of a **Vehicle**.

Why inheritance?

There are clear benefits of using inheritance in our programs. In a sense, it makes the reasoning of an application more possible since it attempts to mimic the world that we live in. But it also reduces code duplication, because similarities between objects can be encapsulated in superclasses. This has the downstream effect of promoting the reuse of code, and intrinsically makes code maintenance much easier. In fact, we often say that if software is not maintained, it dies! So we maintain software often. It behooves us to make this process easier.

Lastly, inheritance makes applications easier to extend. Think of adding a different type of vehicle (say, a **HondaCivic**). Without inheritance, we would have to include instance variables for year, make, model, and so on. However, these are already defined in the **Vehicle** class! We simply need to define the class **HondaCivic** as a subclass of the class **Vehicle** in order to inherit its state and behavior:

```
class HondaCivic(Vehicle):
    ...
```

Single inheritance vs. multiple inheritance

The inheritance relationship that has been discussed thus far is known as **single inheritance**. That is, a subclass inherits state and behavior from a single superclass. Most object-oriented programming languages support single inheritance. Often, however, there is a need to support **multiple inheritance**, where a subclass can inherit from more than one superclass.

To illustrate this, consider a grocery store's items. A banana, for example, is a fruit. Therefore, it may inherit traits from a fruit superclass such as type, country of origin, etc. However, in the context of a grocery store, a banana is also an item for sale. Such a sale item has a price, an inventory, a shelf location, etc. Inheriting from both a **Fruit** superclass and a **SaleItem** superclass, for example, would be useful in implementing the point-of-sale system for a grocery store.

Most object-oriented programming languages do not support multiple inheritance. Some do, but only in a limited form. Java, for example, supports it in a limited form by utilizing something known as an interface. The technical details of this are beyond the scope of this lesson. Python, however, directly supports multiple inheritance. One must merely list all of a subclass' superclasses in the class header; for example:

```
class Banana(Fruit, SaleItem):
    ...
```

OO quick reference

We have covered a lot of new terms in this lesson. This final section merely aggregates them all, along with their definition, so that you can easily and quickly refer to the terms should you need to.

Term	Definition
accessor	A special method in a class that wraps an instance variable for the purpose of providing read access.
behavior	All of the things that an object can *do*; implemented using functions in the class.
class	A blueprint for a thing; the definition of state and behavior for an entire class of things.
class diagram	A diagram that models the classes of a system or application, their relationships, and their members.
class variable	A variable that is defined at the class level; its value is shared among all instances of the class.
constructor	A special method in a class that is automatically invoked when a new instance of the class is instantiated; usually performs initialization tasks (e.g., assigning default or specified values to the instance variables).
decorator	A wrapper; in Python, accessors and mutators are wrapped using a decorator.
dot operator	When used on an object reference, accessed the specified member of the class.
has-a	A relationship among classes that implies one class making use of another; also means the ability of an object to create other objects.
inheritance	A relationship among classes that permits a class to inherit the state and behavior of another class; see **is-a**.
input validation	The process of validating a provided input to ensure that it conforms to some expected range or type.
instance	An object that represents the instantiation of a class.
instance variable	A variable defined in a method of a class (usually the constructor) that allows individual instances of the class to uniquely set values to.
instantiate	The process of constructing a new instance of a class.
is-a	A relationship among classes that permits a class to inherit the state and behavior of another class; see **inheritance**.
magic function	A special function in Python whose name begins and ends with two underscores (e.g., `__init__`, `__str__`, `__add__`).
member	How we collectively reference the state and behavior of a class.
method	How behavior is implemented in a class; they are functions that describe what an object can *do*.
multiple inheritance	The ability of a class to inherit the state and behavior of multiple classes simultaneously.
mutator	A special method in a class that wraps an instance variable for the purpose of providing write access; usually implements input validation.

object	An instance of a class, with specific values assigned to instance variables.
object class	The base class for all user-defined objects; the top-most superclass.
object reference	A variable name that refers to an object.
operator overloading	The redefining of an operator (e.g., the addition operator) on user-defined objects.
single inheritance	The ability of a class to inherit the state and behavior of a single class.
state	All of the things that an object can *be*; implemented using instance variables in the class.
subclass	A class that inherits state and behavior from another class.
superclass	A class that another class inherits state and behavior from.
typecast	The process of converting a value from one type to another (e.g., converting an integer to a floating point number).

The most basic unit of storage is the bit. At any point in time, a bit can be in only one of two states: "0" or "1." Bits are generally implemented as two-state electronic devices (e.g., a current is flowing or not flowing, a voltage is high or low, a magnetic field is polarized in one direction or the opposite direction, etc). The symbol "0" is used to represent one of these states and the symbol "1" is used to represent the other. It really doesn't matter which symbol (the "0" or the "1") represents which physical state (e.g., "high" or "low"). All that is important is that the symbols be assigned consistently and that the two states be clearly distinguishable from each other.

Sequences (or "patterns") of bit values can be used to represent numbers (both positive and negative, integer and real), alphanumeric characters, images, sounds, and even program instructions. In fact, anything that can be stored in a computer must ultimately be stored as a pattern of bit values.

The binary number system

Today, virtually all civilizations use a base ten counting system. However, this has not always been so. In primitive tally systems, for example, there is one stroke for each object being counted. For example, the following tally pattern represents twelve:

Some tally systems group strokes together. The one illustrated above places five strokes in each group. Most early systems attached little or no meaning to the order of the symbols used to represent a number. Roman numerals did use position, but only to indicate whether one value should be added to or subtracted from another value. For example, the Roman numeral MMC stands for 2,100, because "M" represents one thousand, "C" represents one hundred, and the positional rule states that when the symbols are arranged in order of decreasing value, all of the values should be added together. Hence, the meaning of MMC is 1,000 + 1,000 + 100 = 2,100. On the other hand, MCM means 1,900, because the positional rules states that when a symbol for a smaller value immediately precedes a symbol for a larger value, the smaller value is to be subtracted from the larger value. So, MCM is 1,000 + (1,000 – 100) = 1,900. The year 1999 as a Roman numeral is written MCMXCIX, meaning 1,000 + (1,000 – 100) + (100 – 10) + (10 – 1).

Positional notation truly became useful only after the zero digit was introduced. Our modern decimal number system is a base ten positional system. It uses the ten symbols "0" through "9." We count by sequencing through these symbols: "0" for zero, "1" for one, "2" for two, and so on. Once the last symbol is encountered (i.e., "9"), how do we represent the next number? What we need to do is replace the current symbol, "9", with the first symbol in the series, "0", and then increment the symbol immediately to the left of the current symbol by one. Since base ten numbers are assumed to be preceded by (usually unwritten) 0's, the number nine can be written as "09." Hence, cycling "9" back to "0" and incrementing the leftmost "0" to "1" gives "10" as the base ten symbol for the number ten. To continue counting, we cycle the rightmost digit through the symbols "0" through "9" again, producing "10" through "19" for the numbers ten through nineteen. The number twenty can be represented by resetting the "9" to "0" and replacing the "1" with the next symbol in the sequence, "2", giving "20." If we extend this to, say "99," the idea is still the same. The next number, "100," is obtained in the same manner. We first reset the "9" in the right-most digit to "0." We then attempt to increment the next digit

(also “9”), but it, too, is at the end of the sequence. Therefore, we reset it to “0” as well, and increment the left-most “0” to “1,” giving “100.”

Computer systems use base two, or binary, instead of base ten. Counting in binary is similar to counting in base ten. We still cycle through the sequence of symbols, incrementing the symbol to the left of the current symbol whenever the current symbol cycles back to the beginning of the sequence. The only difference is that instead of ten symbols, there are only two symbols: “0” and “1” (hence why it is called the *binary* number system). We begin counting by sequencing through these symbols: “0” for zero, “1” for one, and then we have reached the symbol with the largest value. Keeping in mind that the number one can be rewritten as “01,” we reset the rightmost symbol, “1”, to the first symbol in the sequence, “0”, and then increment the implied “0” immediately to the left to “1” giving “10” (the base two symbol for two).

Note carefully that the symbol “10” (pronounced “one zero”) when interpreted as a base two number refers to the number two, *not* ten. When discussing base two values you should *never* refer to the symbol “10” as “ten” since that is not the value of the number represented by this symbol.

Continuing with the example, the next number, three, can be represented in base two as “11” (we simply increment the right-most digit of “10” from “0” to “1”). To generate the base two representation of four, we begin with three represented as “011” (remember that it is fine to add 0’s to the left-hand side of a number symbol). Next, we set the rightmost “1” digit back to “0” and attempt to increment the middle digit. However, that digit is also at the end of the sequence, since it contains a “1.” So, we reset this digit to “0” as well and proceed to the third (leftmost) digit, which we increment from “0” to “1.” The final result is “100,” which is the base two representation of the number four.

This process for generating base two numbers can be continued indefinitely. The base ten (decimal) and base two (binary) representations of the numbers zero through eight are shown below. For readability, binary numbers are padded to the left with zeros):

Base 10 (decimal)	**Base 2 (binary)**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

Of course, we will need to develop a fast way to find out the number represented by a base two symbol (instead of “counting up to it”). However, before we leave the notion of counting, let’s investigate one other base that is commonly used when discussing programs and data at the machine level.

The hexadecimal number system

Base sixteen, or **hexadecimal**, uses sixteen symbols: "0" through "9" for the numbers zero through nine, and "A" through "F" for the numbers ten through fifteen. The number sixteen is written as "10" in base sixteen, since after the symbol "F" is encountered, it is necessary to cycle back to the beginning of the sequence, "0." When this occurs, the digit immediately to the left of the current digit (an understood "0") is incremented to "1," giving "10." The following illustrates the base ten, base two, and base sixteen representations of the numbers zero through twenty. For readability, binary numbers have been padded to the left with zeros:

Base 10 (decimal)	Base 2 (binary)	Base 16 (hexadecimal)
0	00000	0
1	00001	1
2	00010	2
3	00011	3
4	00100	4
5	00101	5
6	00110	6
7	00111	7
8	01000	8
9	01001	9
10	01010	A
11	01011	B
12	01100	C
13	01101	D
14	01110	E
15	01111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

In order to clearly distinguish which base a number-symbol is to be interpreted under, we generally write the base (two, ten, or sixteen) as a subscript immediately following the digits of the number. Therefore, 11_2 is three, 11_{10} is eleven, and 11_{16} is seventeen. It is standard operating procedure to omit the subscript base if the number is in base 10 (decimal). Repeating the above examples: 11_2 is three, 11 is eleven, and 11_{16} is seventeen.

Number system conversion

Now that we have reviewed the concept of number systems and the idea of counting in a variety of bases, let's look more carefully at what exactly a numeric "base" is. As you learned long ago in grade school, 123 is the way we normally write the number one hundred twenty-three. This is because the "1" is in the *hundreds place*, the "2" is in the *tens place*, and the "3" is in the *ones place*. That is, the digits are positional. Each digit is multiplied by the value of its position (think of this as a weight for each digit position), and the results are then added together. Consider the following way of breaking down the number 123:

10^2	10^1	10^0
100	**10**	**1**
1	2	3

$$= (1 * 10^2) + (2 * 10^1) + (3 * 10^0)$$
$$= (1 * 100) + (2 * 10) + (3 * 1)$$
$$= 100 + 20 + 3$$
$$= 123$$

The system we normally use for representing numbers is called the decimal number system. In this system, the rightmost digit is referred to as being in the one's or "units" position. Immediately to the left of the units position is the ten's position. To the left of the ten's position is the hundred's position, then the thousand's, then the ten thousand's, and so on. The decimal number system is a "base ten" positional number system, because the value of each position can be expressed as a power of the number ten.

The exponent that the base is raised to is given by the position minus one. The right-most position (i.e., position 1), or units position, is 10^0. Note that anything to the power of zero is equal to 1. This right-most position is also known as the least significant position or digit (since it is represented by the lowest power of 10). The tens position is 10^1, the hundreds is 10^2, the thousands is 10^3, and so on. The value of each position is exactly ten times the value of the position immediately to its right.

The other bases work similarly. In the binary number system, the base is two; therefore, the values of the positions (given from right to left) are one (2^0), two (2^1), four (2^2), eight (2^3), sixteen (2^4), thirty-two (2^5), and so on. The value of each position in a base two system is two times the value of the position immediately to its right. For example, the number five is represented in base two as 101_2 (since there is a one in the four's position and a one in the units position). This is illustrated below:

2^2	2^1	2^0
4	**2**	**1**
1	0	1

$$= (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$$
$$= (1 * 4) + (0 * 2) + (1 * 1)$$
$$= 4 + 0 + 1$$
$$= 5$$

In the hexadecimal number system, the base is sixteen. Therefore, the values of the positions (again from right to left) are one (16^0), sixteen (16^1), two hundred fifty-six (16^2), four thousand ninety-six (16^3), and so on. The value of each position in this system is exactly sixteen times the value of the position immediately to its right. The value of the base sixteen number $1A3_{16}$ is four hundred nineteen, since there is one in the two hundred and fifty six's position, ten in the sixteen's position, and three in the units position. This is illustrated below:

16^2	16^1	16^0
256	**16**	**1**
1	A	3

$$= (1 * 16^2) + (A * 16^1) + (3 * 16^0)$$
$$= (1 * 256) + (10 * 16) + (3 * 1)$$
$$= 256 + 160 + 3$$
$$= 419$$

Here's one more example illustrating the representation of the number "nineteen ninety nine" in all three of the bases we have discussed; first, in base ten:

10^3	10^2	10^1	10^0
1000	**100**	**10**	**1**
1	9	9	9

$$= (1 * 10^3) + (9 * 10^2) + (9 * 10^1) + (9 * 10^0)$$
$$= (1 * 1{,}000) + (9 * 100) + (9 * 10) + (9 * 1)$$
$$= 1{,}000 + 900 + 90 + 9$$
$$= 1{,}999$$

Now, in base two:

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1,024	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	1	1	1	0	0	1	1	1	1

$= (1 * 2^{10}) + (1 * 2^{9}) + (1 * 2^{8}) + (1 * 2^{7}) + (1 * 2^{6}) + (0 * 2^{5})$
$+ (0 * 2^{4}) + (1 * 2^{3}) + (1 * 2^{2}) + (1 * 2^{1}) + (1 * 2^{0})$
$= (1 * 1{,}024) + (1 * 512) + (1 * 256) + (1 * 128) + (1 * 64) + (0 * 32)$
$+ (0 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (1 * 1)$
$= 1{,}024 + 512 + 256 + 128 + 64 + 0$
$+ 0 + 8 + 4 + 2 + 1$
$= 1{,}999$

And finally, in base sixteen:

16^2	16^1	16^0
256	**16**	**1**
7	C	F

$= (7 * 16^{2}) + (C * 16^{1}) + (F * 16^{0})$
$= (7 * 256) + (12 * 16) + (15 * 1)$
$= 1{,}792 + 192 + 15$
$= 1{,}999$

One of the most common tasks we face when working with multiple bases is converting numbers from one base to another. We have already seen how to convert from base sixteen and base two to base ten: simply multiply the value of each symbol by the value of its position and add the results together. But how do we convert from base ten to base sixteen or to base two? We also need to figure out how to convert from base two to base sixteen and from base sixteen to base two.

Converting from base two to base sixteen and from base sixteen to base two is easy. In fact, the only reason computer scientists even use base sixteen is because it serves as convenient "shorthand" for base two. The following illustrates the fact that each base sixteen digit can be represented by a group of exactly four base two digits:

Base 16 (hexadecimal)	Base 2 (binary)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110

Base 16 (hexadecimal)	Base 2 (binary)
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

To convert from base sixteen to base two, simply replace each base sixteen digit with its corresponding group of four binary digits. For example, the rather imposing hexadecimal number $1AFC3_{16}$ can easily be converted its binary representation as illustrated below:

1	A	F	C	3
0001	1010	1111	1100	0011

Therefore, $1AFC3_{16} = 1101011111100011_2$. Note that leading zeros have been removed as they are not necessary.

Try to convert $FACE_{16}$ to base two in the table below:

F	A	C	E

Now try to convert $4B1D_{16}$ to base two in the table below:

4	B	1	D

Converting from base two to base sixteen is just as straightforward. We scan the base two number from right to left, replacing each group of four binary digits that we encounter with the equivalent hexadecimal digit. It is important that we group the digits of the base two representation from *right to left*, in case the number of digits is not evenly divisible by four. If this occurs, we simply add leading zeros until the number of digits is divisible by four. Conversion of the bit pattern 11111001111_2 to its hexadecimal representation is shown below:

0111	1100	1111
7	C	F

The original bit pattern, 11111001111, was first broken down from right-to-left into groups of four, and a leading zero was added to the left-most group: 0111 1100 1111. Ultimately, $11111001111_2 = 7CF_{16}$.

Try to convert 1111000000001101_2 to hexadecimal in the table below:

1111	0000	0000	1101

Try to convert $1110111101010010 1_2$ to hexadecimal in the table below:

0001	1101	1110	1010	0101

Note that these conversions between binary and hexadecimal representations in no way change the actual number being represented. For example, $7CF_{16}$ and 11111001111_2 both refer to the same number (1,999), as was illustrated earlier.

We have now looked at conversion methods from base two (and base sixteen) to base ten, from base sixteen to base two, and from base two to base sixteen. The only conversions that we have yet to cover are from base ten to base sixteen and from base ten to base two. We really only need to look at the base ten to base two conversion, since conversion between base two and base sixteen is so trivial. If you have a base ten number and want its base sixteen representation, you can apply a decimal to binary conversion algorithm, and then change the base two result to its base sixteen representation via the grouping method described above.

A number can be converted from decimal to binary by subtracting from it the largest power of two that is less than or equal to the number, and repeating until a remainder of zero is reached. The binary representation of the number is then formed by placing a "1" in the positions corresponding to each of the powers of two that were subtracted. A "0" is placed in the positions corresponding to the powers of two that were not subtracted.

For example, take the decimal number 37. The largest power of two that can be subtracted from it is 32 (2^5), which leaves five. The largest power of two that can be subtracted from 5 is 4 (2^2), which leaves one. Finally, the largest power of two that can be subtracted from 1 is 1 (2^0), which leaves zero. The base two representation of the number is thus formed by placing a "1" in the thirty-two's, four's, and units positions, and by placing a "0" in all other positions. This gives 100101_2. The conversion process for this number is illustrated below:

Number	Minus	Power of 2	Equals	Leftover
37	–	$2^5 = 32$	=	5
5	–	$2^2 = 4$	=	1
1	–	$2^0 = 1$	=	0

2^5	2^4	2^3	2^2	2^1	2^0
32	**16**	**8**	**4**	**2**	**1**
1	0	0	1	0	1

Another method of converting from decimal to binary is to divide the decimal number in half and record both the quotient and the remainder. We then repeat this process with the quotient, while keeping track of the remainder of each division. This is repeated until the quotient is zero. The binary equivalent of the original number is subsequently given by listing the remainders in the reverse order of their derivation (i.e., from the most recent remainder to the first remainder). Here's this method on the decimal number 37:

Dividend	Divided by	Divisor	Equals	Quotient	Remainder
37	/	2	=	18	1
18	/	2	=	9	0
9	/	2	=	4	1
4	/	2	=	2	0
2	/	2	=	1	0
1	/	2	=	0	1

Listing the remainders in reverse order gives 100101 (the same as in the previous example). Therefore, $37_{10} = 100101_2$.

Try to convert the decimal number 642 to base two in the table below:

Dividend	Divided by	Divisor	Equals	Quotient	Remainder
	/		=		
	/		=		
	/		=		
	/		=		
	/		=		
	/		=		
	/		=		
	/		=		
	/		=		
	/		=		

Binary arithmetic

Let's take a look at how arithmetic operations, such as addition and multiplication, can be performed on binary numbers. First, let's examine binary addition of single digit numbers. The simplest case is the addition of 0 plus 0. In binary, it is represented as follows:

$$0 + 0 = 0$$

Some prefer to view this vertically as opposed to horizontally as follows:

$$\begin{array}{rr} & 0 \\ + & 0 \\ \hline & 0 \end{array}$$

Extending this, zero plus one and one plus zero both equal one:

$$0 + 1 = 1$$
$$1 + 0 = 1$$

Or vertically:

$$\begin{array}{rr} & 0 \\ + & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{rr} & 1 \\ + & 0 \\ \hline & 1 \end{array}$$

Finally, one plus one equals two. But the problem is that two cannot be represented as a single binary digit. Instead, we record a zero in the one's position and carry a one over to the two's position. This is represented as follows:

$$1 + 1 = 0 \text{ (carry 1)}$$

Or vertically:

$$\begin{array}{rr} & {\scriptstyle 1} \\ & 1 \\ + & 1 \\ \hline & 0 \end{array}$$

Multi-digit binary addition uses the same strategy employed in decimal addition. One works right-to-left from the least significant digit to the most significant digit, making sure that the carry from the previous column is added to the current column. Because the carry digit for a particular column may be "1" at the same time the corresponding digits of both of the numbers being added are also "1," it is possible to encounter "one plus one plus one equals three" while performing addition. Since 11_2 equals three, "1"should be placed in the current position and another "1" carried over to the position immediately to the left of the current position. This is be represented as follows:

$$1 + 1 + 1 = 1 \text{ (carry 1)}$$

Or vertically:

```
  1
    1
    1
+   1
-----
    1
```

Let's take a look at the addition of 38 + 15 = 53. The following table shows both the binary addition (on the left) and decimal addition (on the right):

	Binary						Decimal
Carry	0	1	1	1	0		1
1st number	1	0	0	1	1	0	38
2nd number			1	1	1	1	15
Sum	1	1	0	1	0	1	53

And now the addition of 43 + 58 = 101:

	Binary							Decimal
Carry	1	1	1	0	1	0		11
1st number		1	0	1	0	1	1	43
2nd number		1	1	1	0	1	0	58
Sum	1	1	0	0	1	0	1	101

Try the addition of 50 + 77 = 127:

	Binary							Decimal
Carry								
1st number								
2nd number								
Sum								

Binary multiplication is also fairly simple. Zero times zero equals zero, as does zero times one and one times zero. One times one equals one. These expressions can be represented in base two as follows:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Notice that none of these four expressions generate a carry, and only one generates a result other than zero. As we will see later, these features lead to binary multiplication being easy to perform; in fact, even easier to perform than decimal multiplication!

Multiplication of multi-digit binary numbers works in a manner similar to multiplication of decimal numbers. As we all learned in grade school, multiplication problems are solved by adding together several partial products. A partial product is formed by multiplying a single digit of the bottom number times the entire top number. For example, given the base ten multiplication problem 472×104, we would solve it in the following way:

$$\begin{array}{cccc} & 4 & 7 & 2 \\ & 1 & 0 & 4 \\ \hline & 1 & 8 & 8 & 8 \\ 4 & 7 & 2 & & \\ \hline 4 & 9 & 0 & 8 & 8 \end{array}$$

The first partial product is given by multiplying 4 times 472, which is 1888. The second partial product is computed as 0 times 472, which is 0. Normally we do not write down zero partial products. The final partial product is 1 times 472. Notice that we write this partial product so that its rightmost digit is directly under the digit of the second number that we multiplied by (i.e., 1). We then add the partial products, column by column from right-to-left, in order to obtain the final answer (49,088 in this case).

We apply this same strategy to perform binary multiplication. Let's take a look at the product of $19 \times 5 = 95$ (in base two: $10011_2 \times 101_2 = 1011111_2$):

	Binary							Decimal
1st number			1	0	0	1	1	19
2nd number					1	0	1	5
Partial products			1	0	0	1	1	
	1	0	0	1	1			
Product	1	0	1	1	1	1	1	95

We form partial products by multiplying the top number by each of the digits of the bottom number. Since the right-most digit of the second number is 1, the first partial product is given by 1 times 10011, or 10011. The right-most digit of this partial product is aligned with the rightmost digit of the second number. We do not record the partial product for zero times something, so multiplying the first number by the second digit of the second number, 0, doesn't generate anything. The final partial product is computed as 1 times 10011 again, but this time where the right-most digit of this result is aligned beneath the third digit of the second number. The partial products are then added to obtain the final result, 1011111_2.

As we have just seen, in binary multiplication the formation of the partial products is very easy since we are only multiplying by 1 (in which case we copy the top number into the proper position) or 0 (in which

case we do nothing). The only *difficult* steps in this process are making sure that we align the partial products correctly and compute the sum of those products accurately.

Now let's try a more difficult problem: the product of $143 \times 23 = 3289$ (in base two: $10001111_2 \times 10111_2 = 110011011001_2$):

	Binary												Decimal
1st number					1	0	0	0	1	1	1	1	143
2nd number								1	0	1	1	1	23
					1	0	0	0	1	1	1	1	429
Partial products				1	0	0	0	1	1	1	1		286
			1	0	0	0	1	1	1	1			
	1	0	0	0	1	1	1	1					
Product	1	1	0	0	1	1	0	1	1	0	0	1	3289

Here, we copy the top number as a partial product everywhere there is a 1 digit in the second number, each time making sure that we align the partial product so that the least significant digit is directly underneath the 1 we are multiplying by. We get the final result by adding the partial products together.

When adding together the partial products, it is important that we handle the carry values properly. Because there is no limit on the size of the numbers to be multiplied, it is possible that there will be a large number of partial products. This situation can lead to carry values that extend over multiple columns. To illustrate this, consider summing a partial product column of five 1s:

Carry	1	0	
Partial products			1
			1
			1
			1
			1
Product			1

Considering these separately, we initially add the first two 1s: $1 + 1 = 10$. We then add the next one to that sum: $10 + 1 = 11$. We then add the next one: $11 + 1 = 100$. Finally, we add the last one: $100 + 1 = 101$. So, $1 + 1 + 1 + 1 + 1 = 101_2$. To record this, we write a 1 in the current column and carry 10, placing the 0 in the column immediately to the left of the current column and 1 immediately to the left of that column. This is no different from the situation we encounter when adding up a long series of decimal numbers. If the current column of digits added to one hundred and one, we would place a 1 in the current column, carry a 0 to the previous column, and carry a 1 to the column before that.

Let's take a closer look at the summation of the partial products of the previous example ($143 \times 23 = 3289$). The right-most column of partial products offers no problem. It is simply 1 plus nothing, giving

a result of 1 with no carry. The second column requires us to add 1 + 1 resulting in a 0 with 1 carried over to the third column. Column three is interesting and is illustrated below:

Carry								1	0	1	0	
				1	0	0	0	1	1	1	1	
			1	0	0	0	1	1	1	1		
		1	0	0	0	1	1	1	1			
1	0	0	0	1	1	1	1					
									0	0	1	

The sum of the digits in column three, including the carry, is $1 + 1 + 1 + 1 = 100_2 = 4$. Hence, a 0 is written in column three, a 0 is carried to column four, and a 1 is carried to column five.

Column four contains three 1s, giving us a sum of 11_2. Hence, we write a 1 in column four and carry a 1 to column five. Note that the 1 we just carried to column five joins the carry of 1 already in that column:

Carry								1				
								1	0	1	0	
Partial products					1	0	0	0	1	1	1	1
				1	0	0	0	1	1	1	1	
			1	0	0	0	1	1	1	1		
	1	0	0	0	1	1	1	1				
Product									1	0	0	1

Column five now contains a total of five 1s (including the two carries). Since five is written in binary as 101_2, we write a 1 in column five, and carry a 0 into column six and a 1 into column seven. The current state of the summation of partial products after adding the contents of column five is illustrated below:

Carry								1				
						1	0	1	0	1	0	
Partial products					1	0	0	0	1	1	1	1
				1	0	0	0	1	1	1	1	
			1	0	0	0	1	1	1	1		
	1	0	0	0	1	1	1	1				
Product								1	1	0	0	1

The remainder of the computation is carried out in a similar manner, always being careful to handle the carries properly. Try it out in the table below:

Carry								1				
						1	0	1	0	1	0	
Partial products					1	0	0	0	1	1	1	1
				1	0	0	0	1	1	1	1	
			1	0	0	0	1	1	1	1		
	1	0	0	0	1	1	1	1				
Product								1	1	0	0	1

Binary adder

An adder, as its name implies, is a circuit for adding binary numbers. The simplest adder adds two bits. As shown above, adding two bits can result in the following:

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} {\scriptstyle 1}\ \\ 1 \\ +\ 1 \\ \hline 10 \end{array}$$

When adding, two parts are produced: a sum and a carry (each of which can be either 0 or 1). A circuit to implement the behavior of an adder will need two inputs (one for each of the single-bit numbers) and two outputs (one for the sum and one for the carry). Constructing such a circuit is fairly straightforward. Consider the following truth table for the adder (where S is sum and C is carry):

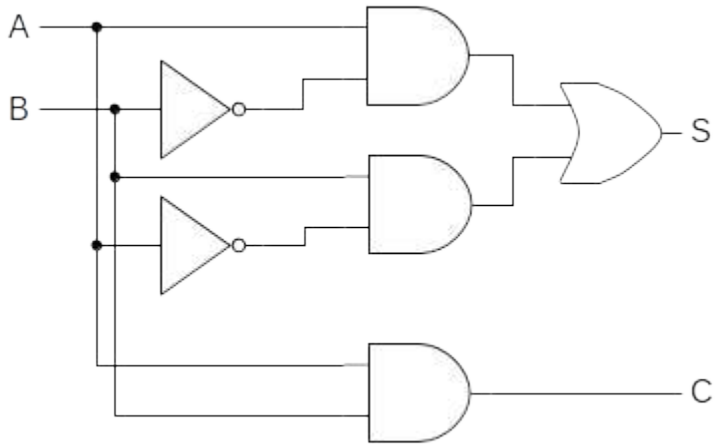
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Notice that the values in the S column correspond to an *xor* of the two inputs, while the values in the C column correspond to an *and* of the two inputs. Here are their respective truth tables for reference:

XOR		
A	**B**	**Z**
0	0	0
0	1	1
1	0	1
1	1	0

AND		
A	**B**	**Z**
0	0	0
0	1	0
1	0	0
1	1	1

Note how the output of the *xor* gate is exactly the same as the sum bit produced by the adder. Similarly, the output of the *and* gate is exactly the same as the carry bit produced by the adder. Constructing the circuit is almost too easy:



This circuit is called a **half adder**. It has two Boolean expressions: $S=(A\cdot\overline{B})+(\overline{A}\cdot B)$ and $C=A\cdot B$. While a half adder does add two single-bit numbers and can generate a carry, it has no provision for a carry *input* into the circuit. As shown above, when adding two multi-bit binary numbers, one works column by column from right-to-left, making sure that the carry bit from the previous column is added into the current column. Here is the illustration of this process show earlier, on 38 and 15:

	Binary						Decimal
Carry	0	1	1	1	0		1
1st number	1	0	0	1	1	0	38
2nd number			1	1	1	1	15
Sum	1	1	0	1	0	1	53

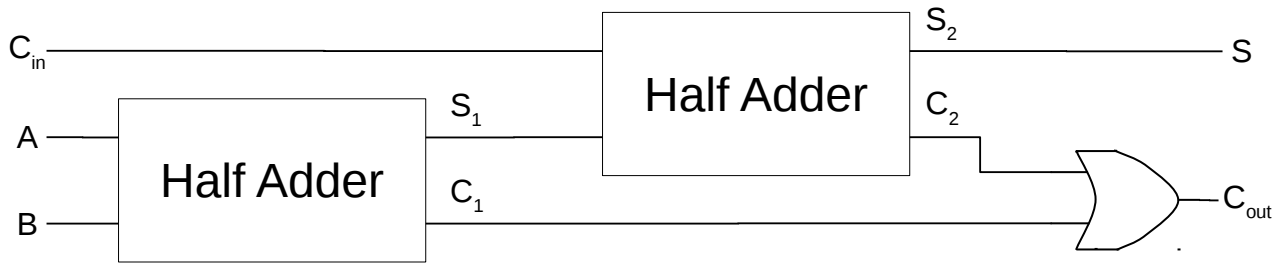
A half adder could be used to add the right-most (low-order) bits of the two numbers, but it is not general enough to add the digits of an arbitrary column since it does not support a carry as input.

A **full adder** overcomes this limitation of the half adder by allowing a carry to be fed into the circuit along with a bit from each of the numbers to be added. Thus, a full adder will have three inputs: the two bits being added, plus a *carry in*. Only two output bits, the sum and a *carry out*, are needed because the largest result that can be produced by the circuit will be three (11_2). This occurs when all three inputs are 1. Here is a complete truth table for a full adder. The inputs are almost the same as before, except that the carry in is labeled C_{in}, and the carry out is labeled C_{out}:

C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Designing a circuit to implement this behavior from scratch would be challenging. However, through careful observation and the use of two half adders, the job is manageable. Since we'll be using half adders to build the full adder, our circuit diagram will be simpler if we imagine the half adder encapsulated into a *black box*, as shown below:

The trick to building a full adder is to think of the sum of the three terms, $A + B + C_{in}$, as a sequence of two sums associated left to right: $(A + B) + C_{in}$. One half adder will be used to compute the sum $A + B$. The *sum* bit output by this half adder, along with C_{in}, will be fed as input into a second half adder. The *sum* bit produced by the second half adder will serve as the *sum* bit of the full adder. The *carry out* bit of the full adder is produced by routing the *carry out* bits of both half adders into an *or* gate. This is illustrated below:

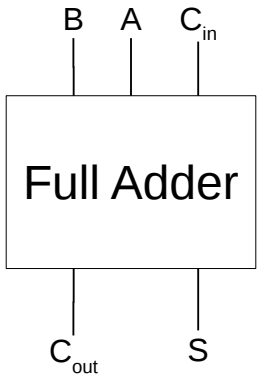


Try to develop a complete implementation of the full adder using only *and*, *or*, and *not* gates below:

Now verify that this circuit does, in fact, generate the truth table for binary addition below:

C_{in}	A	B	S	C_{out}

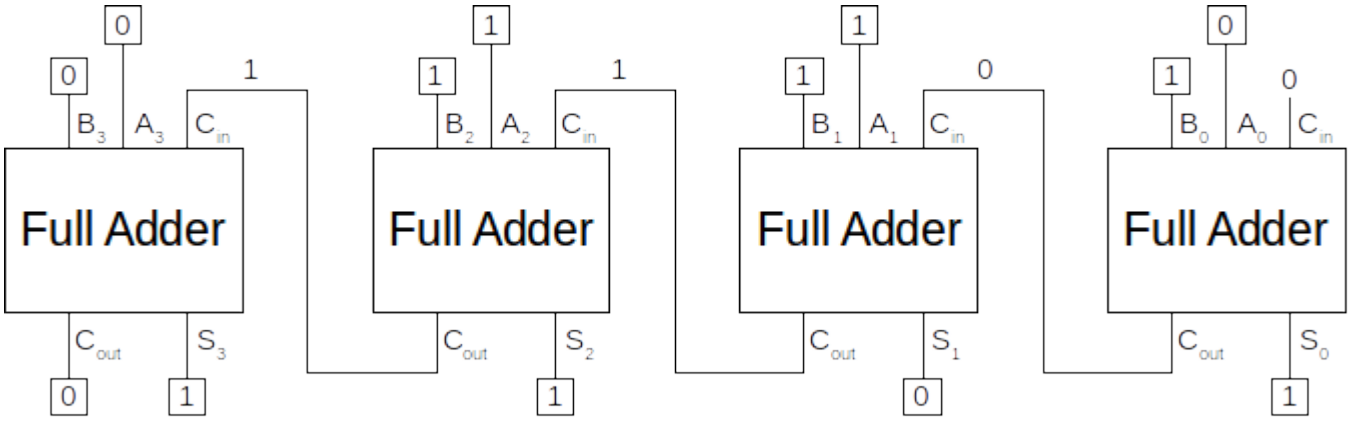
Just as we encapsulated the half adder, we can encapsulate the single-bit full adder into a black box.



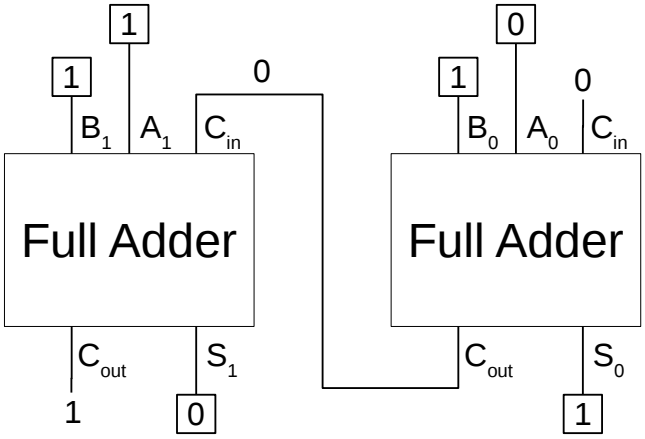
This representation looks a little different than the previous circuit because it has been rotated clockwise 90 degrees to make the following figure easier to read. Note that the inputs, outputs, behavior, and internal details of the circuit remain unchanged.

Multi-bit adders can be implemented as a chain of single-bit full adders where the *carry out* of each adder is routed to the *carry in* of the adder immediately to its left. Under this scheme, each full adder is essentially responsible for adding a single bit of each of the two input numbers, plus the carry bit generated by the adder immediately to its right. The *carry in* for the rightmost adder is permanently set to 0. The *carry out* of the leftmost adder indicates whether or not addition of the inputs produces an overflow. We will cover why this is necessary later when discussing how numbers are represented.

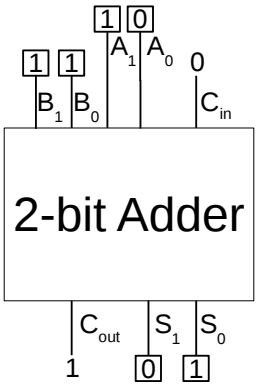
Here's a four-bit adder constructed from four single-bit (full) adders. In this example, A holds the number six (0110_2) and B holds seven (0111_2). The result of this addition operation is 1101_2, or thirteen:



Interestingly, we can continue to build this. To illustrate this, let's go back one step to the full adder. We can chain two full adders together (each of which can effectively produce the sum and C_{out} of A, B, and C_{in}) to produce the sum of two 2-bit numbers as follows:

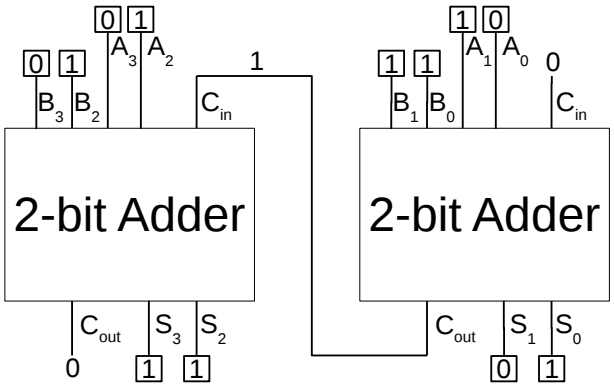


Indeed, $10_2 + 11_2 = 101_2$ (the overflow bit is 1). And now we can box the two full adders into a single 2-bit adder as follows:



The result is the same as chaining two full adders. The C_{out} of the first full adder that is wired to the C_{in} of the second full adder is now internal to the 2-bit adder. This 2-bit adder effectively adds two 2-bit numbers (A and B composed of the bits A_0, A_1, B_0, and B_1). It produces two sums (S_0 and S_1 – one for each bit) and a C_{out} (the overflow bit).

We can chain several 2-bit adders in a similar manner to produce a 4-bit adder as follows:



The effect is the same as show earlier when we chained four full adders. This combination of two 2-bit adders produces the same sum: $0110_2 + 0111_2 = 1101_2$ (the overflow bit is 0).

Of course, this can be continued. We could encapsulate the two 2-bit adders into a 4-bit adder, and chain two of those to produce a single 8-bit adder. Two 8-bit adders could be encapsulated into a single 16-bit adder. Two 16-bit adders could be encapsulated into a single 32-bit adder. And this can go on and on. The incredible part about this is that, in the end, a seemingly complicated 32-bit adder is still just made up of many full adders chained together, which are themselves made up of half adders, which are themselves made up of the three primitive logic gates: *and*, *or*, and *not*. Simply amazing!

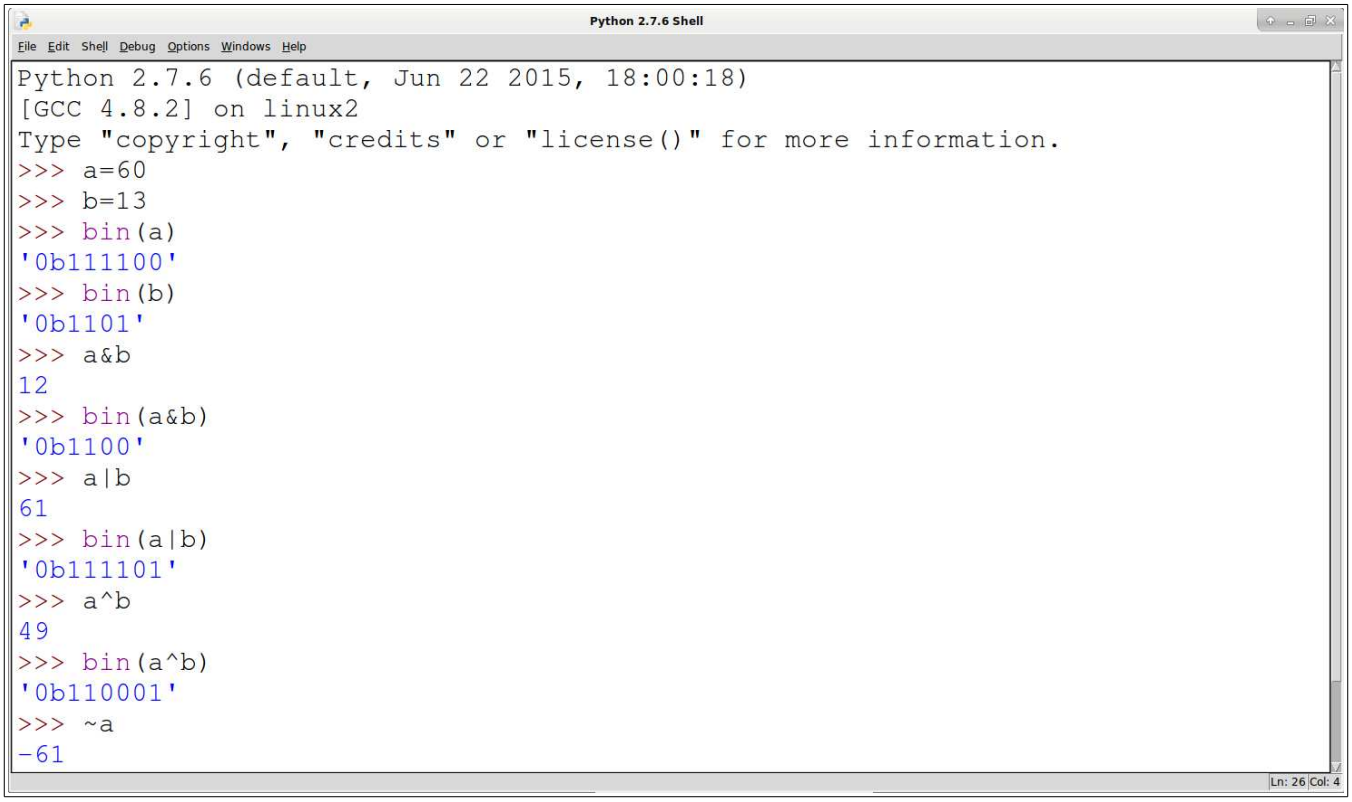
Bitwise operators in Python

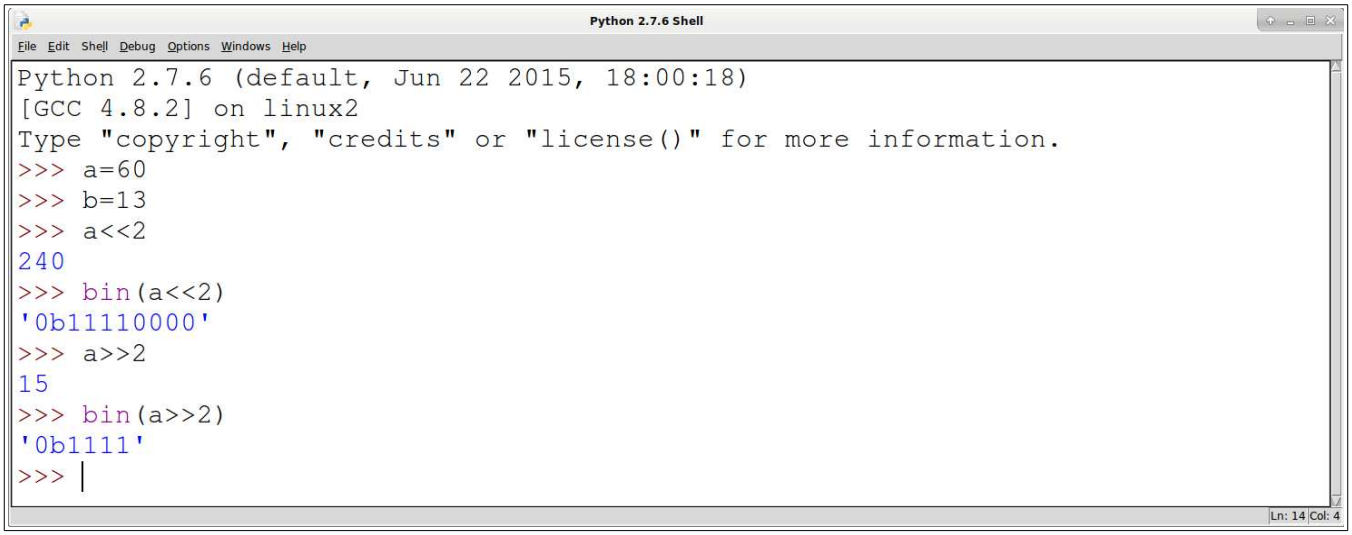
Up to this point, we've left out one final class of operators in Python: the **bitwise operators**. The reason is that they really only make sense once we understand how numbers are represented in computers. Moreover, how binary arithmetic works is fundamental to understanding them.

The **bitwise operators** work on bits and perform bit-by-bit operations. Think back to the primitive logic gates (*and*, *or*, and *not*) and some derivatives (e.g., *xor*). Each of these concepts operated on bits and produced bits. In the following table, assume that a = 60 (or 00111100 in binary) and b = 13 (or 00001101 in binary):

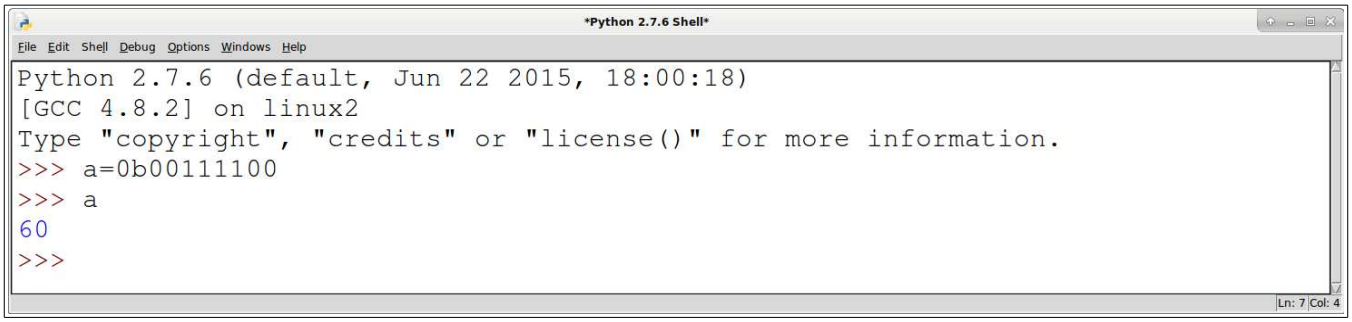
Python Bitwise Operators and Examples		
&	bitwise *and*	a & b = 00001100 (or 12 in decimal)
|	bitwise *or*	a | b = 00111101 (or 61 in decimal)
^	bitwise *xor*	a ^ b = 00110001 (or 49 in decimal)
~	bitwise *not*	~a = 11000011 (or -61 in decimal; we will explain this one later)
<<	left shift	a << 2 = 11110000 (or 240 in decimal)
>>	right shift	a >> 2 = 1111 (or 15 in decimal)

The bitwise not has the effect of inverting the bits. Why 11000011 in binary is equal to -61 in decimal will be explained in a later lesson. Here is output of the examples in the previous table in IDLE:

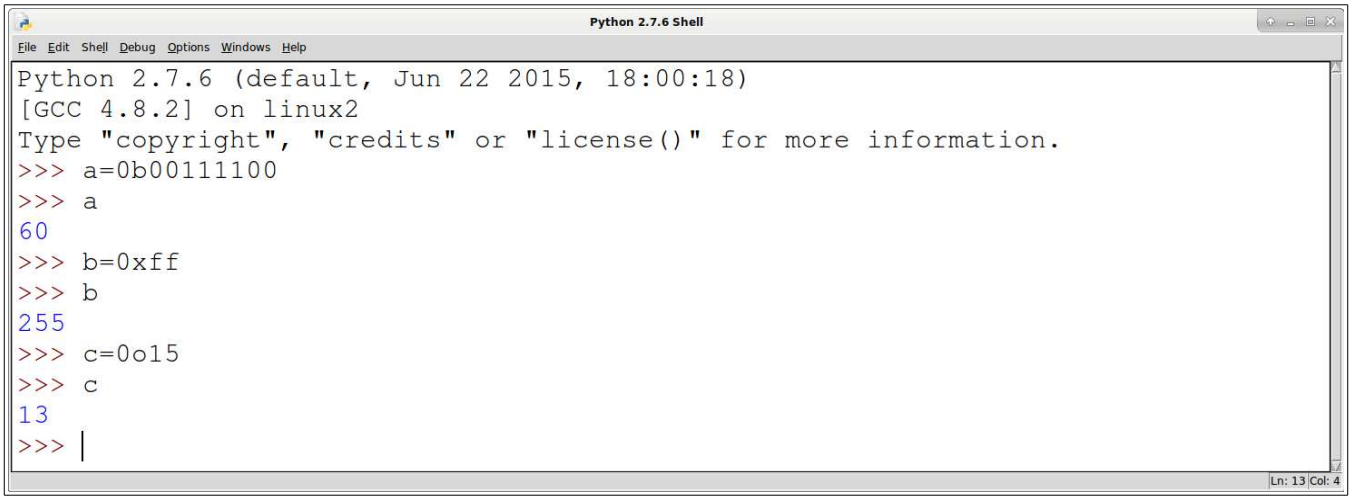




Note the use of the **bin** function. It returns the binary representation of a value. If a = 60, the statement `bin(a)` returns 0b111100 (which is 60 in binary). The prefix 0b implies binary. In fact, you can assign values to variables in binary form using this prefix:



This can be done in other bases as well. For example, in hexadecimal (with the prefix 0x) or in octal (with the prefix 0o):



Graphical User Interfaces Beam

Computing devices are indeed ubiquitous. We use them all the time for a variety of tasks. To use them, we must inherently interact with them. In fact, there's an entire branch of computer science that deals with the way humans interact with machines (primarily computing machines) called human-computer interaction (HCI). In this lesson, we will focus on the way that we typically interact with computing devices: through graphical user interfaces.

A **graphic#l user interface** (GUI – pronounced "gooey") is a type of interface that allows users to interact with computing machines through graphical entities. An **interface** is just a program that provides a way of interacting with a computing machine so that a user can use it to do its bidding! The interface typically monitors input (e.g., clicking with the mouse, typing on the keyboard, etc) and controls output (e.g., the result of inputs rendered to a monitor). GUIs are the most common type of interface between humans and computers. In fact, most software products and virtually all popular computer operating systems are GUI-based.

Formally, GUIs implement an interface metaphor, such as a *desktop*, that allows users to interact with computer systems. Software objects, such as programs and data files, are generally represented as postage stamp sized pictures called **icons**. A human can access one of these objects by selecting it with a pointing device, such as a **mouse**. When *opened*, an icon expands to become a **window**. A window is a portion of the computer screen used to communicate with a particular program. Generally, windows and icons can be resized (i.e., made larger or smaller) and repositioned anywhere on the desktop. But of course, you already know all this!

GUI components
Although the types of components that make up a GUI can vary depending on operating system, application, and various other factors, there are many standard ones that virtually all GUIs support:

- **Window**: an area on the screen that displays information.
- **Menu**: allows users to execute commands by selecting from a list of options.
- **Icon**: a small picture that represents something else (e.g., a file, an application, a command).
- **Control**: also known as a **widget**, a component that users directly interact with (e.g., by clicking, dragging, etc) to perform some task (e.g., launch an application, set a configuration setting, etc). There are many types of controls that you may be familiar with: list, label, check box, radio button, slider, spinner, and so on.
- **Tab**: a way of grouping GUI components in an area of a window.

Events
On their own, GUIs really do nothing. That is, they are specifically designed for user interaction. Users must interact with a GUI, and when that happens, an event occurs that *triggers* some sort of action. An **action** can be virtually anything, such as opening a file, launching an application, performing a background task, and so on. We often say that some GUI components *listen* for user interaction; that is, they implement what is known as a **listener**. When the listener detects user interaction (e.g., through a mouse click), the interaction is registered as an **event**. The event then triggers some predefined action that handles it and typically produces some output that is expected by the user.

The Python Tkinter library
By default, Python does not support GUIs. That is, Python is text-based. In fact, the majority of programming languages are text-based. The print statement, for example, displays text to the console (the terminal on the RPi, for example). It does not, for example, display text to a graphical window or on a label positioned somewhere on the user's desktop. The integration and support of GUIs in programming languages is typically done through internal or external libraries that provide the objects and tools necessary to create GUIs and to allow users to interact with GUIs.

Although there are many GUI libraries that work with Python, this lesson will focus on one of the more common and popular ones that is included with Python by default and is cross-platform (i.e., it can be used to create GUIs on a variety of computing machines and operating systems). This GUI library is called **Tkinter**[1].

The Tkinter library is complex, yet it is powerful enough to create just about any GUI that a programmer would need. It would not be possible to cover the entire library in a single lesson. Therefore, we will take the approach of introducing the most frequently used components that Tkinter supports. Subsequently, we will show how to create a few simple GUIs that demonstrate a few of the different types that can be created.

A Tkinter primer
Tkinter stands for "Tk interface." Tk is a GUI toolkit that has been around for a while and was originally developed by the same folks that created the Java programming language. Tkinter is an object-oriented *layer* that provides a Python interface to the Tk GUI toolkit.

The first thing we must do to use the Tkinter library in a Python program is to import it via:

```
from Tkinter import *
```

Although there are various ways to import the library, the above is the most common method of doing so that reduces the amount of source code required.

The typical manner in which GUIs are created using the Tkinter library is to create a window on which other components are placed. Let's look at a simple program that creates a window and places a label on it:

```
1: from Tkinter import *

2: window = Tk()
3: text = Label(window, text="GUIs in Python are pretty easy!")
4: text.pack()
5: window.mainloop()
```

Here's the output of this short program:

[1] For more information on Tkinter, see https://wiki.python.org/moin/TkInter.

Let's explain the program, line-by-line. You already know the statement on line 1 that imports the Tkinter library. The statement on line 2 creates a graphical window (that is stored in the variable `window`). This must always be done in order to create a GUI using the Tkinter library. Line 3 creates a Tkinter **Label**, which is a component that is used to display text, an icon, or an image on a GUI. The label, which has the text "GUIs in Python are pretty easy!", is stored in the variable `text` and is *bound* (or attached) to the window as its *child*. These explain the parameters passed in to the constructor of the **Label** class defined in the Tkinter library. The `pack` function called on line 4 is used on all Tkinter widgets. In this case, it instructs the label to size itself to fit the specified text and to make itself visible on the window. However, the GUI is not actually shown on the desktop until the statement on line 5. This statement instructs the window to appear on the desktop and wait for the user to interact with it. It will remain on the desktop until the user closes the window (in this case, by clicking on the X at the top-right of the window).

Let's try a more standard way of creating GUIs using the Tkinter library. This method uses the object-oriented paradigm to encapsulate the GUI in a class that inherits from a Tkinter **Frame**. Here's the program[2]:

```
 1:  from Tkinter import *

 2:  class App(Frame):
 3:       def __init__(self, master):
 4:            Frame.__init__(self, master)
 5:            self.button1 = Button(master, text="BYE!",\
 6:                 fg="red", command=self.quit)
 7:            self.button1.pack(side=LEFT)
 8:            self.button2 = Button(master, text=\
 9:                 "Say something!", command=self.say)
10:            self.button2.pack(side=LEFT)

11:       def say(self):
12:            print "Froot Loops!"

13:  window = Tk()
14:  app = App(window)
15:  window.mainloop()
```

Here's the resulting GUI:

Clicking on the BYE! button closes the application. Clicking on the Say something! button displays the string "Froot Loops!" to the console each time it is clicked.

Let's explain the statements of the program as we did before, in the order that they are executed. Clearly, line 1 imports the Tkinter library. Lines 2 through 12 define the **App** class and are not yet

[2] Inspired by an example from the Tkinter tutorial: An Introduction to Tkinter.

executed. Line 13 creates the main window of the GUI. Line 14 creates a new instance of the **App** class, passing the main window as a parameter (it becomes the parent of any GUI components created in the **App** class). This launches the constructor of the **App** class which begins on line 3. Since the **App** class is a subclass of the Tkinter **Frame** class, the constructor of the **Frame** class is first called on line 4. This initializes a Tkinter **Frame**, which serves as a holder for other GUI components. A Tkinter **Button** (called `button1`) is then instantiated as an instance variable on lines 5 and 6. It is set as a child of the window (called `master` in the **App** class), given the text "BYE!" colored in red, and instructed to execute the function `quit` when clicked. The function `quit` is defined in the Tkinter library. For a frame, it simply closes it.

Line 7 makes the button visible. Note the parameter: `side=LEFT`. This places the button as far left as possible in the frame. The default is `TOP`, which places a component as far to the top as possible. A second button, `button2`, is instantiated in lines 8 and 9. This button is instructed to execute the function `say` when clicked, which is defined in the **App** class (on lines 11 and 12). The button is also positioned as far left as possible (next to `button1`). The function `say` simply displays the text "Froot Loops!" to the console.

Finally, line 15 displays the GUI and allows the user to interact with it.

Common Tkinter widgets
Before we go on to create more elaborate GUIs, let's discuss a few of the more common Tkinter (in general, GUI) widgets. In fact, the Tkinter library supports fifteen core widgets:

- **Button**: a button that can be used to execute an action when clicked.
- **Canvas**: used to draw graphs, plot points, create drawings, etc.
- **Checkbutton**: a button that can represent two distinct values by being checked or unchecked.
- **Entry**: used to provide text-based user input.
- **Frame**: a container that can group other widgets.
- **Label**: used to display text or an image.
- **Listbox**: used to display a list of options that the user can select from.
- **Menu**: used to implement pull-down menus by grouping menu items.
- **Menubutton**: a single menu item that is used in pull-down menus.
- **Message**: like a Label, used to display text; however, it is more configurable.
- **Radiobutton**: a button in a group of buttons that represents one of the values associated with the group.
- **Scale**: supports the selection of a numeric value by dragging a *slider*.
- **Scrollbar**: provides horizontal and vertical scroll bars for various GUI components.
- **Text**: supports formatted text, including embedded images and even windows.
- **Toplevel**: a container that can be displayed as a separate window on top of other components.

Discussing all of these widgets in detail and showing how they can be used in GUIs is beyond the scope of this lesson. If you wish, you can visit many online tutorials to see these widgets (and more) used in the creation of GUIs.

Configuring widgets
Most Tkinter widgets can be configured as they are instantiated by specifying various parameters in the constructor. Here's an example with the Button widget shown above:

```
b = Button(master, text="Submit", fg="blue", bg="yellow")
```

Another way of configuring a widget is to invoke its `config` function. In general, a widget, *w*, can be configured as follows:

```
w.config(option=value, option=value, ...)
```

For example:

```
b.config(text="Send", fg="red")
```

Positioning widgets

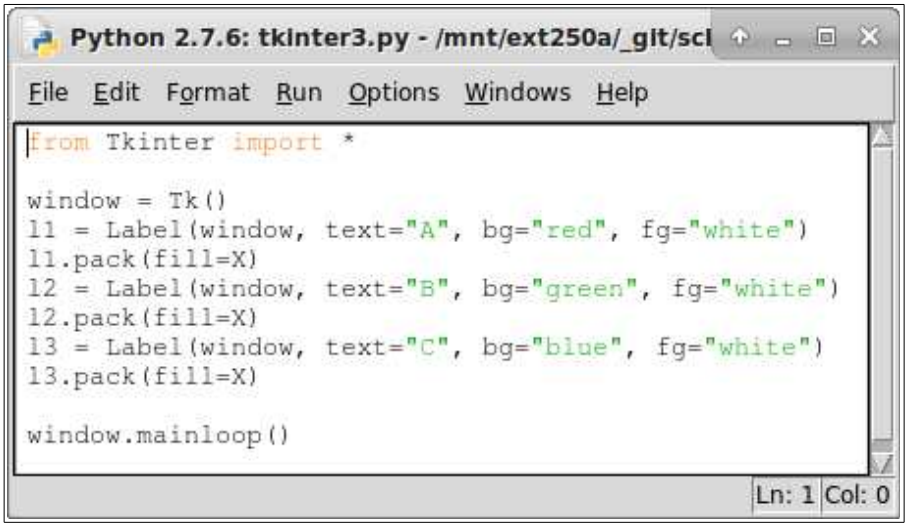
Widgets can be positioned in the main window or in a frame using a variety of layouts. The one that has been used in previous examples is called the **pack manager**. It configures widgets in rows and columns. Options such as *fill*, *expand*, and *side*, help determine where a widget is placed and how it behaves graphically. The pack manager is good for placing a single widget and having it fill an entire container. It is also good for placing widgets next to each other vertically or horizontally.

The pack manager's *fill* option is used to have a widget fill the entire space assigned to it. There are several values that can be assigned to this option: BOTH makes the widget expand both horizontally and vertically; X makes the widget expand only horizontally; and Y makes the widget expand only vertically.

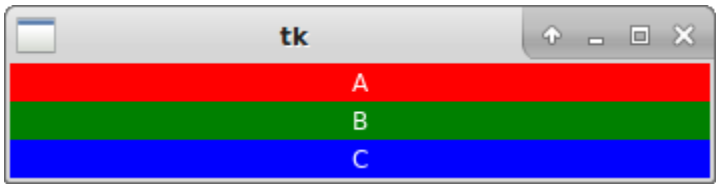
The pack manager's *expand* option is used to assign any additional space in a container to a widget. That is, if the parent container has any remaining space after packing all widgets, it will be distributed among all widgets that have the *expand* option set to a non-zero value.

The pack manager's *side* option is used to specify which side of the container to place the widget against. Values for this option are: TOP (the default) which packs widgets vertically; LEFT which packs widgets horizontally; BOTTOM which packs a widget against the bottom; and RIGHT which packs a widget to the right. Note that, although these values can be mixed, the results may not be as intended. Of course, you can easily experiment!

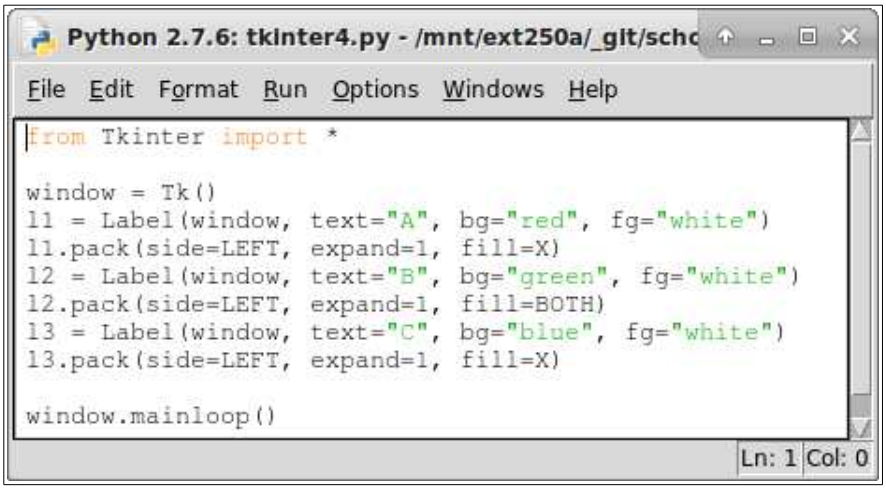
Here's an example of aligning Tkinter **Label**s vertically, while expanding them horizontally:



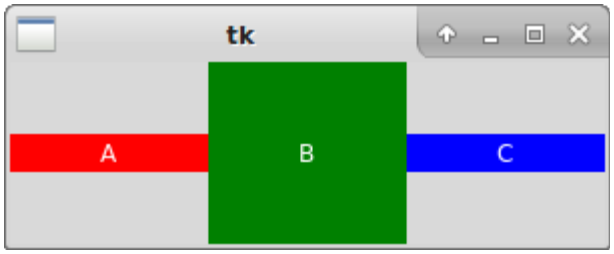
And here's the output (note that the window was manually resized to be larger horizontally):



And here's a GUI aligning the labels horizontally, expanding them horizontally, and expanding only the middle one vertically:



And here's the output (again, the window was manually resized):



There is a much more powerful and flexible layout manager in the Tkinter library called the **grid manager**. It also configures widgets in rows and columns; however, each widget's row and column (and how it behaves in its position in the row and column) can be individually specified. For the purpose of this lesson, the row and column in which a widget is placed is known as a **cell**. The grid manager supports several options: *row*, *column*, *sticky*, *columnspan*, and *rowspan*.

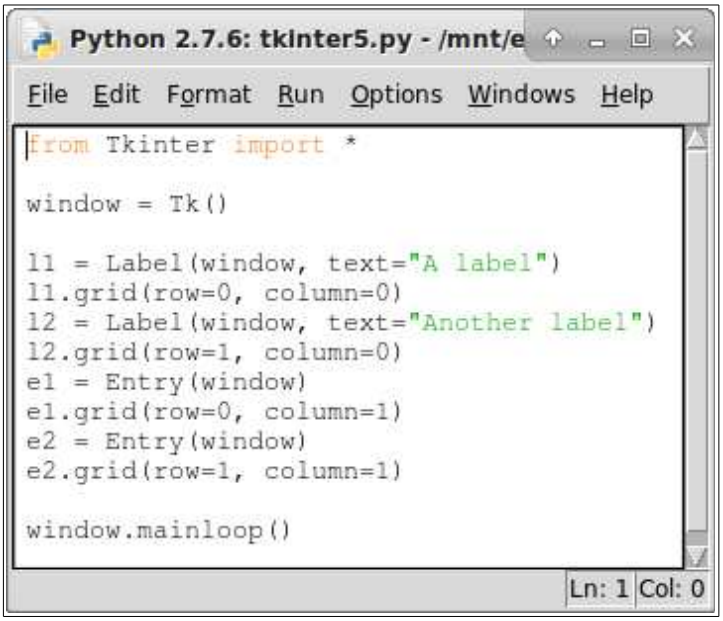
The *row* option is a numeric value that specifies which row in the grid the widget should be placed in. Rows begin at 0. The *column* option is a numeric value that specifies which column in the grid the widget should be placed in. Similarly, columns begin at 0.

The *sticky* option aligns the widget based on several values: N aligns the widget to the North (i.e., the top); S aligns the widget to the South (bottom); E aligns the widget to the East (right); and W aligns the widget to the West (left). Values can be combined; for example, NE aligns the widget to the Northeast

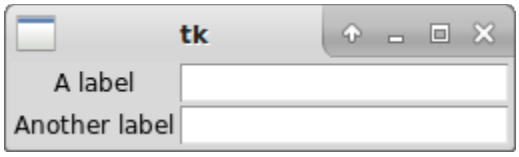
(top-right). They can also be stacked; for example, N+S expands the widget vertically, E+W expands the widget horizontally, and N+S+E+W expands the widget both vertically and horizontally.

The *columnspan* option allows a widget to span multiple columns. Similarly, the *rowspan* option allows a widget to span multiple rows.

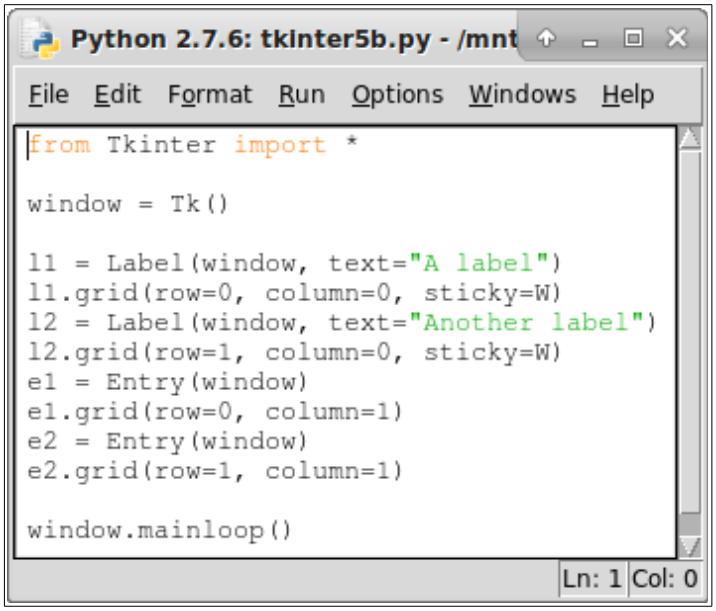
Here's a first example using the grid manager:



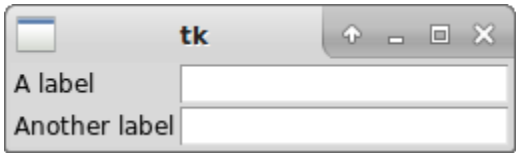
The program creates two labels and two text entry fields. Here's its output:



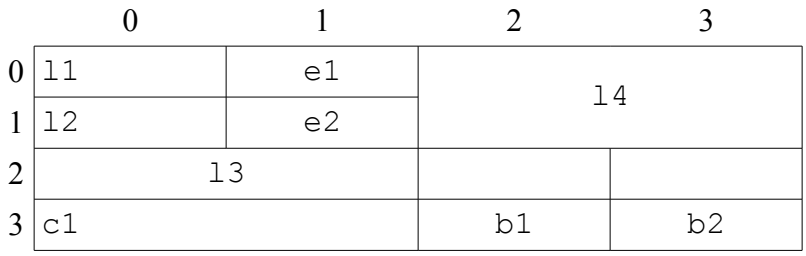
Notice how the labels are, by default, aligned in the center of their cell. We can force them to be aligned to the left by slightly modifying the source code as follows:



Note how the *sticky* option has been added to the two labels. Here's the output of the modified program:



Let's add more widgets to see how the rest of the grid manager options can be used. First, however, it is good practice to doodle by drawing what it is that we are trying to accomplish, especially for GUIs that are complicated (typically, that's when there are more than just a few widgets). Here's a quick mock-up of what the GUI will look like:

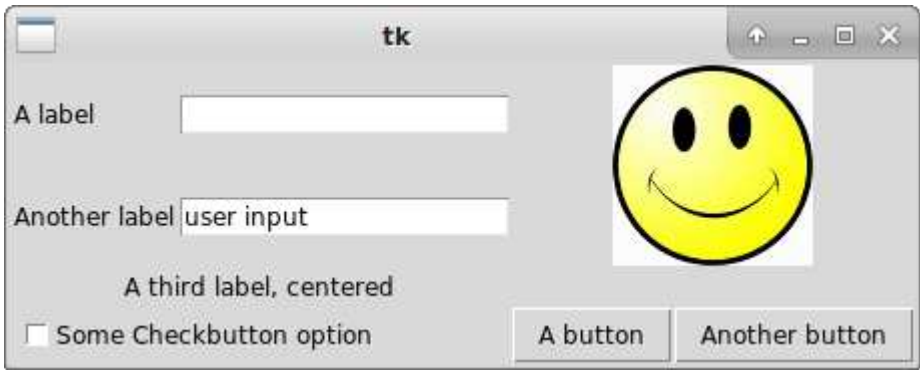


For clarity, here are the widget variable names, and their types and descriptions:

- `l1`: a **Label** with the text "A label", left-aligned in row 0, column 0.
- `l2`: a **Label** with the text "Another label", left-aligned in row 1, column 0.
- `l3`: a **Label** with the text "A third label, centered", centered horizontally in row 2, spanning across columns 0 and 1.
- `l4`: a **Label** with a "smiley" image (100x100 pixels), centered horizontally and vertically, spanning across rows 0 and 1, and columns 2 and 3.

- `e1`: an empty **Entry**, centered horizontally in row 0, column 1.
- `e2`: an **Entry** with the text "user input", centered horizontally in row 1, column 1.
- `c1`: a **Checkbutton** with the text "Some Checkbutton option", left-aligned in row 3, spanning across columns 0 and 1.
- `b1`: a **Button** with the text "A button", centered horizontally in row 3, column 2.
- `b2`: a **Button** with the text "Another button", centered horizontally in row 3, column 3.

Before we get to the source code that creates this GUI, let's take a look at the end result:



Now, let's build the code to create this GUI, a little at a time. We'll start with the following:

```
from Tkinter import *

class GUITest(Frame):
   def __init__(self, master):
        ...

   def setupGUI(self):
        ...

window = Tk()
t = GUITest(window)
t.setupGUI()
window.mainloop()
```

At this point, all that's been done is to create the main window. Note that we will be implementing the GUI as a class (called **GUITest**) that inherits from the Tkinter **Frame** class. That is, it's just a frame on which other widgets will be placed. The **GUITest** class will, of course, have a constructor. We'll also implement a `setupGUI` function that does the bulk of instantiating and positioning the widgets. This explains the statement `t.setupGUI()` in the main part of the program at the bottom. The process is to first create the main window, then create the instance of the **GUITest** class (which is a frame), then invoke its `setupGUI` function to create the GUI, and finally to display the GUI with the statement `window.mainloop()`.

Let's work on the constructor of the **GUITest** class:

```
        def __init__(self, master):
            Frame.__init__(self, master)
```

```
        self.master = master
```

The constructor first calls the constructor of its superclass (the **Frame** class). Then, it declares an instance variable, `master`, that stores the main window. This is necessary so that the `setupGUI` function can add widgets as children of the main window.

Now on to the `setupGUI` function. We'll build the function a little at a time. First, let's add the first label, `l1`:

```
    def setupGUI(self):
        l1 = Label(self.master, text="A label")
        l1.grid(row=0, column=0, sticky=W)
```

The first statement instantiates a new **Label**, makes it a child of the main window (again, called `master` in the **GUITest** class), and sets its text. The second statement defines its properties with respect to the grid manager. It is to be positioned in row 0, column 0, and is to be left-aligned (to the West).

The next label, `l2`, is similarly created; however, it is to be positioned in row 1, column 0, and has different text:

```
        l2 = Label(self.master, text="Another label")
        l2.grid(row=1, column=0, sticky=W)
```

The third label is centered across two columns (0 and 1; therefore, it spans across two columns) in row 2. It also sets the sticky option to E+W, meaning that it will evenly split any leftover space within its container to the left and right (i.e., it will be centered):

```
        l3 = Label(self.master, text="A third label, centered")
        l3.grid(row=2, column=0, columnspan=2, sticky=E+W)
```

The fourth and final label is an image that is centered across two rows (0 and 1) and two columns (2 and 3). Images are handled a bit differently than text. The image must first be loaded from a file and stored in a variable. This is done by using Tkinter's **PhotoImage** class:

```
        img = PhotoImage(file="smile.gif")
```

Note that the specified image file must be located in the same directory as the Python program. The label can then be created, with the image set as its contents:

```
        l4 = Label(self.master, image=img)
        l4.image = img
        l4.grid(row=0, column=2, columnspan=2, rowspan=2,\
            sticky=N+S+E+W)
```

The second statement appears to be redundant. That is, the first seems to assign the variable `img` as the label's image; however, the second statement seems to do the same thing. It turns out that an image created using the **PhotoImage** class is "garbage collected" when a function that created it terminates. Once the `setupGUI` function terminates, the image disappears from the GUI. To prevent this from happening, we can keep an extra reference to the image. The second statement does this. The sticky option is set to N+S+E+W, which centers the image both horizontally and vertically.

Next, the first text entry is created, centered in row 0, column 1:

```
e1 = Entry(self.master)
e1.grid(row=0, column=1)
```

As mentioned earlier, widgets are centered by default when positioned using the grid manager. The second text entry widget is similarly added; however, it is in row 1, column 1. In addition, it contains default text that is added by using the `insert` function in the **Entry** class. This is accomplished by inserting text at the END position of the text entry widget:

```
e2 = Entry(self.master)
e2.insert(END, "user input")
e2.grid(row=1, column=1)
```

The **Checkbutton** widget is left-aligned in row 3 and spans across columns 0 and 1:

```
c1 = Checkbutton(self.master,\
     text="Some Checkbutton option")
c1.grid(row=3, column=0, columnspan=2, sticky=W)
```

Finally, the two buttons are added. The first button is centered in row 3, column 2:

```
b1 = Button(self.master, text="A button")
b1.grid(row=3, column=2)
```

And the second button is centered in row 3, column 3:

```
b2 = Button(self.master, text="Another button")
b2.grid(row=3, column=3)
```

This completes the `setupGUI` function. Here's a complete listing of the function for reference:

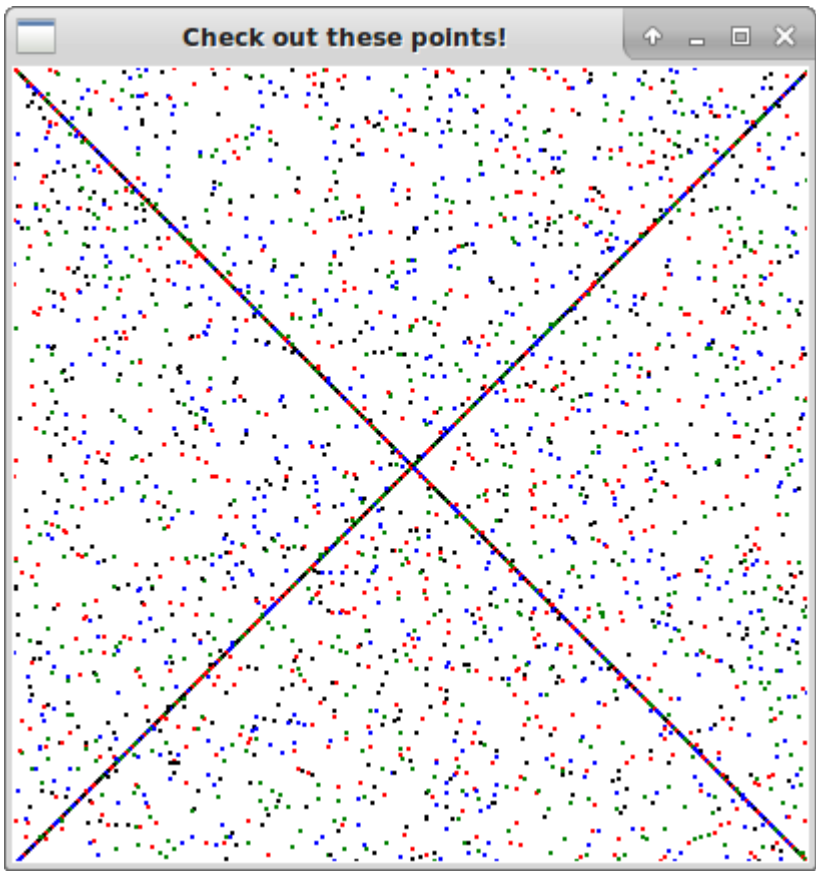
Python 2.7.6: tkinter5c.py - /mnt/ext250a/_git/school/teaching/Spring2016/csc132

File Edit Format Run Options Windows Help

```
from Tkinter import *

class GUITest(Frame):
        def __init__(self, master):
                Frame.__init__(self, master)
                self.master = master

        def setupGUI(self):
                l1 = Label(self.master, text="A label")
                l1.grid(row=0, column=0, sticky=W)

                l2 = Label(self.master, text="Another label")
                l2.grid(row=1, column=0, sticky=W)

                l3 = Label(self.master, text="A third label, centered")
                l3.grid(row=2, column=0, columnspan=2, sticky=E+W)

                img = PhotoImage(file="smile.gif")
                l4 = Label(self.master, image=img)
                l4.image = img
                l4.grid(row=0, column=2, columnspan=2, rowspan=2, sticky=N+S+E+W)

                e1 = Entry(self.master)
                e1.grid(row=0, column=1)

                e2 = Entry(self.master)
                e2.insert(END, "user input")
                e2.grid(row=1, column=1)

                c1 = Checkbutton(self.master, text="Some Checkbutton option")
                c1.grid(row=3, column=0, columnspan=2, sticky=W)

                b1 = Button(self.master, text="A button")
                b1.grid(row=3, column=2)

                b2 = Button(self.master, text="Another button")
                b2.grid(row=3, column=3)

window = Tk()
t = GUITest(window)
t.setupGUI()
window.mainloop()
```

Ln: 1 Col: 0

As a final example, let's create a GUI on which two-dimensional points are plotted in various colors as follows:



The **CanvW** widget allows plotting points, among other useful things. It is quite important to note that the top-left of the canvas is the origin with the coordinates (0,0). It is possible to shift the origin, but this is beyond the scope of this lesson.

Continuing with the strategy of implementing a class that inherits from a Tkinter widget, we'll structure this example such that our user-defined class, **Points**, inherits from Tkinter's **Canvas** class. Since some of the points that will be plotted are random (that is, with random *x*- and *y*-coordinates), the `randint` function of Python's **random** class will be used. For flexibility, several constants will be defined: the width and height of the window, the various colors that points can be plotted in, the radius of the points, and the total number of points to plot. Structuring the program in this way will make it very easy to quickly test the program with different parameters.

Let's take a look at the source code:

Python 2.7.6: tkinter6.py - /mnt/ext250a/_git/school/teaching/Spring2016/csc132/03 Beam 1 - Graph

File Edit Format Run Options Windows Help

```
from Tkinter import *
from random import randint

WIDTH = 400
HEIGHT = 400
POINT_COLORS = ["black", "red", "green", "blue"]
POINT_RADIUS = 0
NUM_POINTS = 2500

class Points(Canvas):
        def __init__(self, master):
                Canvas.__init__(self, master, bg="white")
                self.pack(fill=BOTH, expand=1)

        def plotPoints(self, n):
                for i in range(WIDTH):
                        self.plot(i, i)
                        self.plot(WIDTH - i - 1, i)
                for i in range(n):
                        x = randint(0, WIDTH - 1)
                        y = randint(0, HEIGHT - 1)
                        self.plot(x, y)

        def plot(self, x, y):
                color = POINT_COLORS[randint(0, len(POINT_COLORS) - 1)]
                self.create_oval(x, y, x + POINT_RADIUS * 2, y + POINT_RADIUS * 2, outline=color)

window = Tk()
window.geometry("{}x{}".format(WIDTH, HEIGHT))
window.title("Check out these points!")
p = Points(window)
p.plotPoints(NUM_POINTS)
window.mainloop()
```

Ln: 1 Col: 0

By default, the width and height of the main window is defined to be 400x400 pixels. Knowing that the top-left of the canvas is the origin, then the bottom-right must have the coordinates (399,399). The points are plotted in four colors: black, red, green, and blue. Using the **Canvas** class, points are drawn as ovals. An oval is specified by a rectangular bounding box (specifically by its top-left and bottom-right coordinates). If these coordinates are the same, the the oval is just a point. You will see how the constant `POINT_RADIUS` will be used in a later example. Finally, 2,500 points are plotted.

The main part of the program (at the bottom of the source code) should be familiar. There are, however, two new statements. The first sets the width and height of the window (collectively known as the window's geometry):

```
window.geometry("{}x{}".format(WIDTH, HEIGHT))
```

This statement sets the window's dimensions to the values of the constants `WIDTH` and `HEIGHT`. The second statement sets the window's title (the text at the top of the window):

```
window.title("Check out these points!")
```

The final three statements create the GUI, plot the points, and display the GUI on the desktop.

Let's take a look at the **Points** class:

```
 1:  class Points(Canvas):
 2:       def __init__(self, master):
 3:            Canvas.__init__(self, master, bg="white")
 4:            self.pack(fill=BOTH, expand=1)

 5:       def plotPoints(self, n):
 6:            for i in range(WIDTH):
 7:                 self.plot(i, i)
 8:                 self.plot(WIDTH - i - 1, i)
 9:            for i in range(n):
10:                 x = randint(0, WIDTH - 1)
11:                 y = randint(0, HEIGHT - 1)
12:                 self.plot(x, y)

13:       def plot(self, x, y):
14:            color = POINT_COLORS[randint(0,\
15:                 len(POINT_COLORS) - 1)]
16:            self.create_oval(x, y, x + POINT_RADIUS * 2,\
17:                 y + POINT_RADIUS * 2, outline=color)
```

The constructor of the **Points** class first calls the constructor of its superclass, Tkinter's **Canvas** class. Note that the constructor can take various configuration parameters. In this case, the background is additionally set to white. The canvas is then configured to expand horizontally and vertically to fill the main window (i.e., it will expand to fit in the 400x400 pixel window).

The `plotPoints` function does the bulk of generating the points to plot. The first part of the code (in lines 6 through 8) generates the set of points required to produce an X shape in the GUI. The variable *i* iterates from 0 through 399 (all valid *x*- and *y*- coordinate locations on the canvas). The first set of points generated have matching coordinates (e.g., (0,0), (1,1), (2,2), and so on). This forms the set of points from the top-left to the bottom-right of the canvas. The second set of points generated have opposite coordinates with respect to the width of the canvas. That is, when the *x*-component is 0, the *y*-component is 399; when the *x*-component is 1, the *y*-component is 398; and so on. This forms the set of points from the bottom-left to the top-right of the canvas.

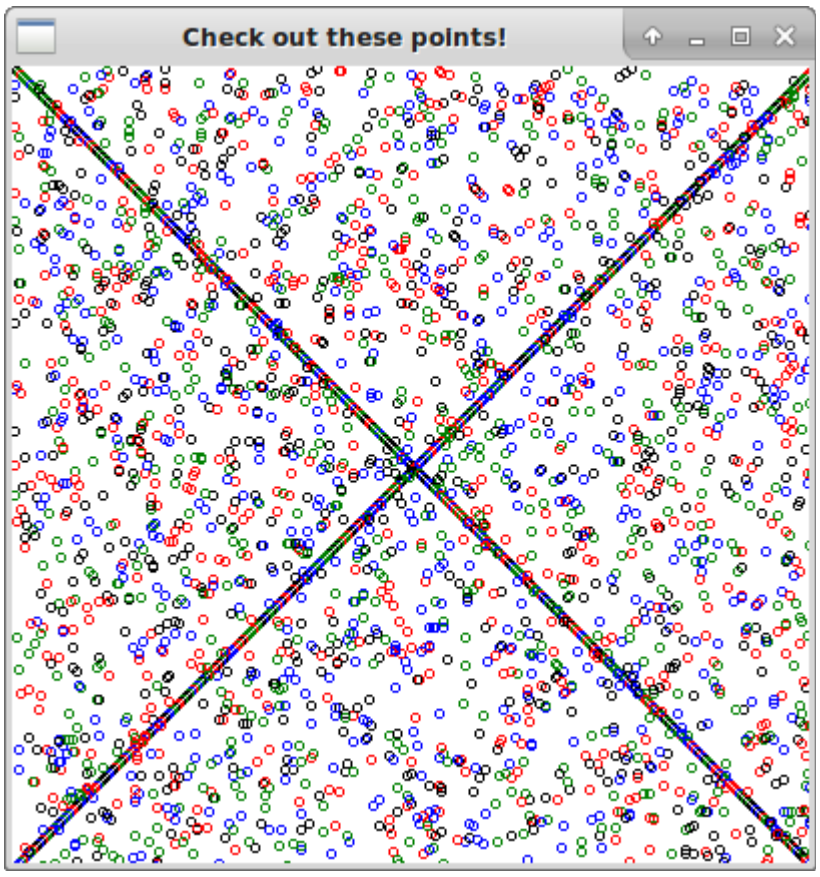
The second part of the `plotPoints` function (in lines 9 through 12) generates *n* random points (2,500 in the case of the program above). Recall that the `randint` function takes a closed interval. That is, the first and last parameters passed in to the function express the valid range of random integers to generate, inclusive of the low and high values.

Finally, the `plot` function (in lines 13 through 17) actually plots the points on the canvas. First, in lines 14 and 15, a color is randomly selected from the list of colors that was defined as a constant near the top of the program. Next, an oval is drawn on the canvas within the specified bounding box. With the current set of parameters, the oval is just a point with no radius. The oval is drawn in the randomly selected color by specifying its outline to be of that color.

We can observe the behavior of the `create_oval` function in the **CanvW** class a bit more by changing the radius of the points to something larger (say, 2) by modifying the `POINT_RADIUS` constant at the top of the program as follows:

```
POINT_RADIUS = 2
```

The output of the program looks a bit different:



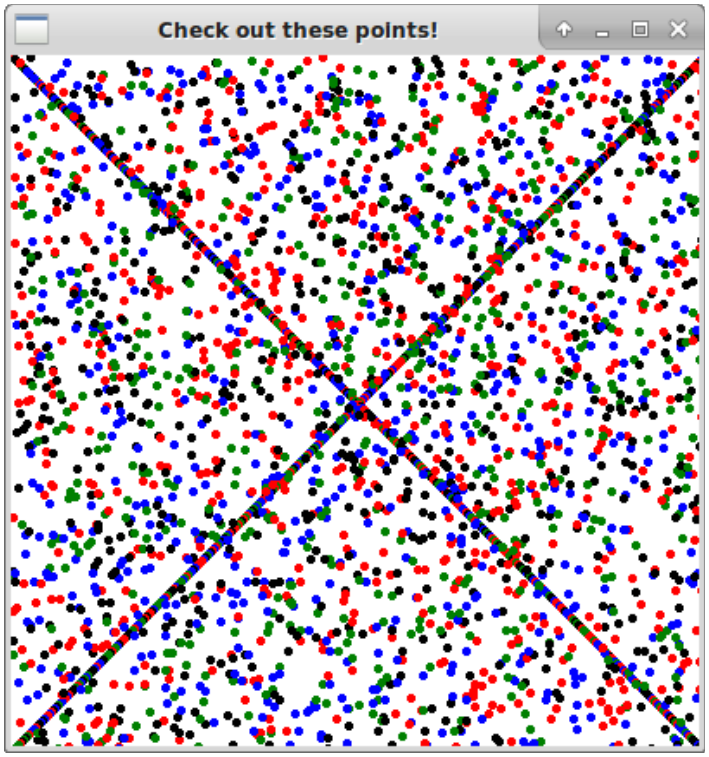
The "points" now have a radius that is greater than 0. The bounding box of a "point" is defined by the *x*- and *y*-coordinates of the point and the point's radius. The top-left of the bounding box is just the *x*- and *y*-coordinates of the point. The bottom-right takes the radius into account. That is, the coordinates of the point really represent its top-left "corner", while its bottom-right "corner" is adjusted to account for the specified radius (twice, in fact, to make up the entire diameter). When we keep the radius the same for both corners of the bounding box, the oval becomes a circle. In fact, a circle is just a special case of an oval.

We can actually fill the points as well by slightly modifying the statement on lines 16 and 17:

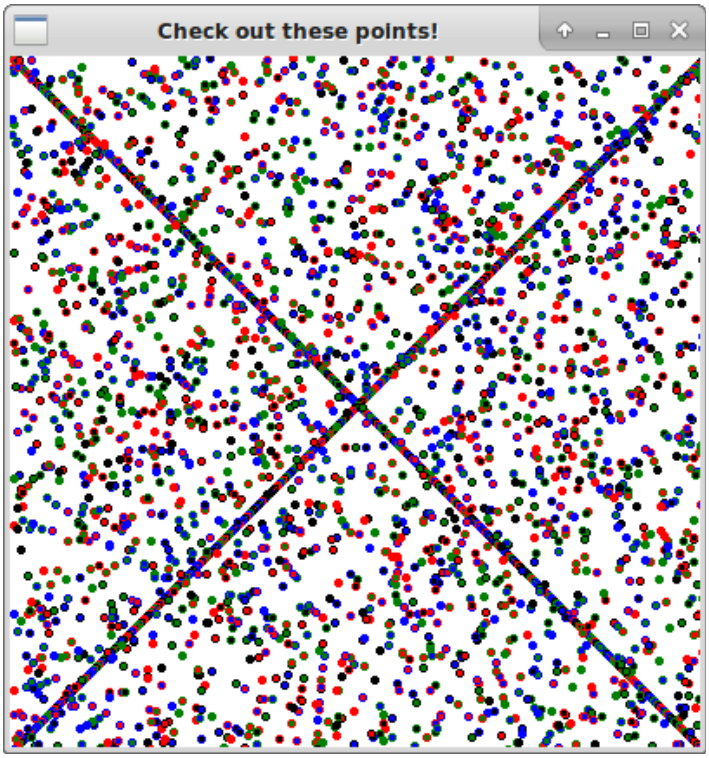
```
16:  self.create_oval(x, y, x + POINT_RADIUS * 2,\
17:        y + POINT_RADIUS * 2, outline=color, fill=color)
```

We include a *fill* option that takes a color that the center of the oval will be filled with. Clearly, an oval can have two different colors: one for its outline, and one for its "inside".

Here's the output of the program with the *fill* option set as specified above:



Lastly, here's the output of the program with the *fill* option set to a random color, and the *outline* option set to another random color (note that it is possible that both colors are the same):

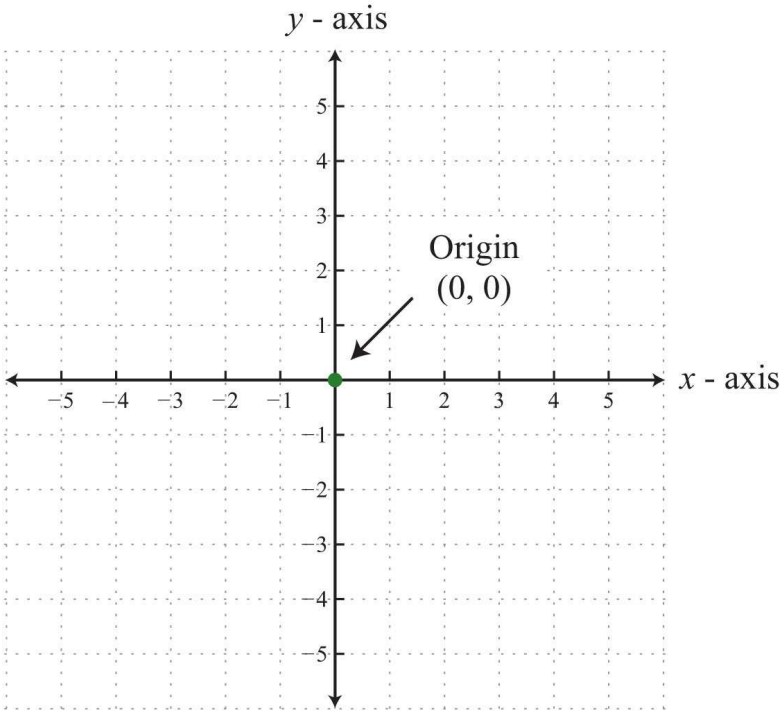


In this lesson, we are going to study chaos and order. While they intuitively mean the complete opposite of each other, we will find out that there is in fact a very close relationship between the two. Often, if you look closely enough at something apparently chaotic, you may find that there is some order to it after all.

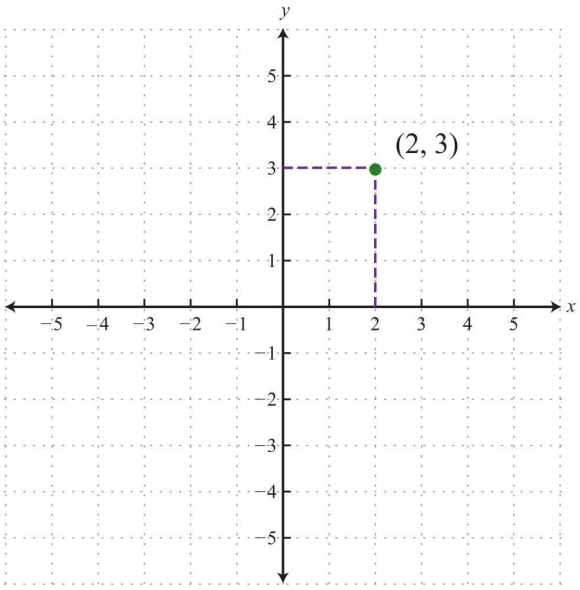
The coordinate system

A **coordinate system** allows us to represent positions of a specified dimension in a reproducible fashion. This means that if someone describes a position to us using a coordinate system, we can accurately represent that position. For example, the two-dimensional coordinate system allows us to represent points on a flat plane (similar to the page of a book or a white board on a wall). The system is made up of two axes that are perpendicular (i.e., at 90 degrees) to each other, and which meet at a point referred to as the **origin**. The position of any point on that plane is described using two values that describe how far away from the origin the given point is along both predefined axes. We typically refer to these axes as *X* and *Y*, with the *X*-axis oriented horizontally (i.e., left-to-right), and the *Y*-axis oriented vertically (i.e., up-to-down). Although this represents a way to locate points in two dimensions, we can extend this to more dimensions. For example, we can add an axis oriented at an angle (technically, you can think of it as being perpendicular to both the *X*- and *Y*-axes and going through the page or the white board). This adds a third dimension to the coordinate system and allows the location of objects in three-dimensional space.

Each axis in a coordinate system has values along it that demarcate one-dimensional positions. Although we have freedom in what those values are, we typically center the origin at the X-value 0 and the Y-value 0. We call this a point. The origin, then, rests at point (0, 0). The values along an axis typically increase or decrease by 1. Here's an example of a two-dimensional coordinate system:

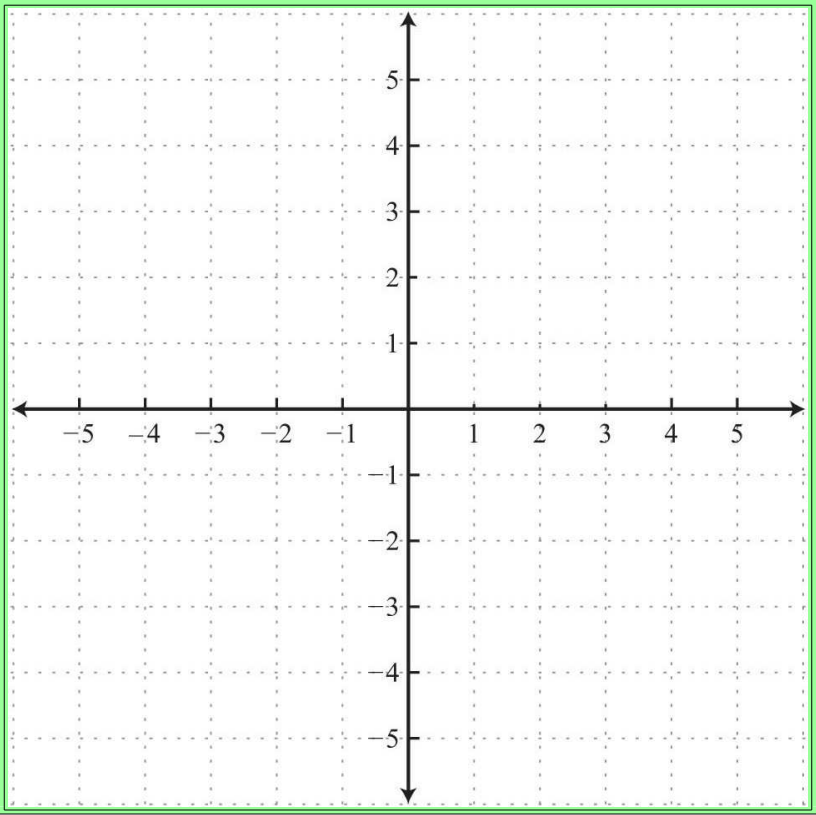


When specifying points on the coordinate system, we typically list the X-component first, followed by the Y-component. That is, the point (2, 3), for example, has an X-value of 2 and a Y-value of 3. We could plot this point as follows:



Activity 1: Plotting points

Can you place these five points in their proper positions on the coordinate system below: (3, 5), (5, -2), (1, 2), (-3, -5), and (-4, 0)?



Another benefit of using the coordinate system for describing the position of different points is that it allows us to easily calculate and plot a point in the middle of two points. We call such a point the midpoint of the two points. If two points were drawn on a plain piece of paper, and we were asked to mark the point in the exact middle of those two points, it would be a long and (many times) inaccurate process. However, having the coordinates of the two points makes this simple. The midpoint of two points is calculated by adding the corresponding coordinates and dividing by two to find the average of each component. For example, the midpoint of the points (5, -2) and (1, 2) is given by the following coordinates:

$$\left(\frac{5+1}{2}, \frac{-2+2}{2}\right) = (3, 0)$$

That is, the x- and y-components of each point are averaged to find the x- and y-components of the midpoint. Similarly, the midpoint of the points (-3, -5) and (7, 3) is given by the following coordinates:

$$\left(\frac{-3+7}{2}, \frac{-5+3}{2}\right) = (2, -1)$$

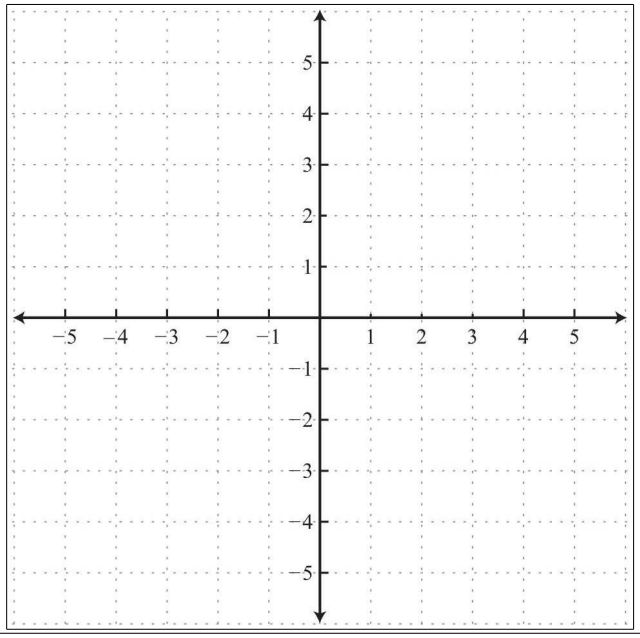
Try to calculate the midpoints of the following point combinations in the space below:

Points (0, 0) and (4, 4):

Points (-3, 5) and (-1, -3):

Points (4, -4) and (-4, -4):

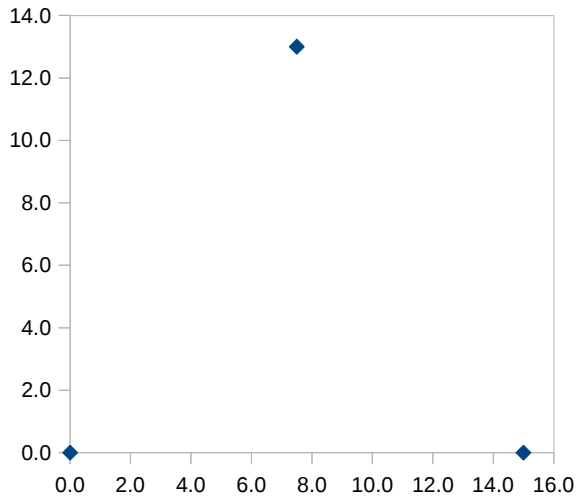
Now plot the given points and the calculated midpoints below:



The chaos game

The chaos game is an interesting (and, hopefully, fun) game that illustrates how seemingly random and chaotic things can produce something orderly (and beautiful) over time. It works in a simple manner. First, plot the three vertices of an equilateral triangle. An equilateral triangle is unique in that each of its three sides are the same length, and each of its three vertices are equidistant to each other. Since the internal angles of a triangle always sum to 180 degrees, each angle of an equilateral triangle is the same (i.e., each angle is 60 degrees).

Here is an example of the vertices of an (almost) equilateral triangle:



Notice that the three vertices of the equilateral triangle are drawn on a coordinate system at the points (0,0), (15,0) and (7.5,13). Although these points are not "perfectly" equidistant, they are nearly so (and good enough for this example).

Second, randomly select two of the three vertices. Suppose that the top and bottom-right vertices are randomly selected. Calculate the midpoint between these two vertices using the midpoint formula as illustrated above. In general, given two points (x_1, y_1) and (x_2, y_2), the midpoint can be calculated as follows:

$$\left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$$

The midpoint of the top and bottom-right vertices at points (7.5, 13) and (15, 0) is calculated as follows:

$$\left(\frac{7.5+15}{2}, \frac{13+0}{2}\right) = \left(\frac{22.5}{2}, \frac{13}{2}\right) = (11.25,6.5)$$

The midpoint, (11.25, 6.5), is then plotted on the coordinate system.

The game now follows a repeatable pattern. The calculated midpoint at point (11.25, 6.5) is selected, along with one randomly selected vertex of the equilateral triangle. Suppose that the top vertex at point (7.5, 13) is randomly selected. A new midpoint is now calculated:

$$\left(\frac{11.25+7.5}{2}, \frac{6.5+13}{2}\right) = \left(\frac{18.75}{2}, \frac{19.5}{2}\right) = (9.375, 9.75)$$

this new midpoint is then plotted on the coordinate system, and the process continues (i.e., the latest midpoint is selected, along with a randomly selected vertex of the equilateral triangle). Here is the algorithm in pseudocode:

```
v1, v2, v3 ← three vertices of an equilateral triangle
plot v1, v2, and v3
p1, p2 ← two randomly selected vertices from v1, v2, and v3
m ← the midpoint of p1 and p2
plot m
repeat
    v ← a randomly selected vertex from v1, v2, and v3
    m' ← the midpoint of m and v
    plot m'
    m ← m'
until midpoints plotted >= 1500 or tired
```

By playing the chaos game for some time, what do you think will happen? Where do you think the points will focus on the coordinate system? Will they collect in any one place at all?

Activity 2: The chaos game...manually

Let's play this game a little bit to see if anything can be gleaned (i.e., if anything *reveals* itself). Mainly, we want to see if the points just randomly fill the coordinate system, or if they somehow produce something *orderly*.

To obtain results quickly, let's use blank sheets of paper (or even better, transparencies). Everyone should plot the exact same vertices of an equilateral triangle. From there, play the chaos game and plot approximate midpoints. That is, estimate as best as possible (by looking) where the midpoint should be plotted. Remember that, to plot the first midpoint, two random vertices are selected. From that point on, the last plotted midpoint and one randomly selected vertex is used to compute the next midpoint.

Each student should repeat this process on his/her own individual paper or transparency at least 25 times. It's best to plot at least 100 midpoints. The paper or transparency can then be combined with those of the other students to see what (if anything) is formed. If using transparencies, simply lay them on top of each other, aligning the vertices. The result should be clearly visible. If using paper, the procedure is the same; however, you will need to hold the stack of paper to the light (note that it may be difficult to see all of the points).

So what is the result?

Did you know?

For any pattern to be revealed, a single person would have to plot thousands of points (which would take a really long time). There is a whole branch of computer science that deals with making computers faster, particularly for large tasks. One of the techniques that addresses this involves dividing tasks so that small sub-tasks are done concurrently (i.e., at the same time). This field is called h**igh performance computing**, and the technique employed is called **parallelism**.

Activity 3: The chaos game...automated

As you can see, it takes a lot of points to plot anything reasonable. That is, in order to actually see if anything orderly is produced when playing the chaos game, a lot of midpoints must be calculated and plotted.

There are also a lot of other interesting things that can be tried. For example, what would happen if the three vertices were not equidistant; that is, what if the triangle was different? What if we plotted a square instead (i.e., four vertices instead of three)? What about a pentagon? What would happen if, instead of plotting midpoints, points at varying distances were plotted instead. For example, would the result be different if a three-fourths point was plotted instead each time (i.e., instead of 50% of the distance to the randomly selected vertex, 75% of the distance). What about 10% of the distance? What would happen if there was an additional rotation about the randomly selected vertex? That is, what would be the result if some midpoint (or other distance) were calculated and then rotated about the vertex some number of degrees (some angle)?

These are all interesting questions that may result in absolute randomness (or perhaps not!). It would be quite difficult to try these manually. Why not automate this process? A frequent task of computing professionals is to automate things that tend to be repeated. To illustrate a number of changes to the chaos game in a configurable manner, a simple application has been created to test out all of the options.

The goal of this activity is to play around with this application to see how changes to the chaos game affect the outcome. Try your hand at it!

Fractals

By now, you should have noticed that the result of the original chaos game (i.e., three vertices of an equilateral and repeatedly plotting midpoints) results in a beautiful triangular shape. In fact, this shape has an interesting property: it has its pattern repeated over and over. It's as if we could zoom in forever and obtain the same pattern! This property is found in fractals. This particular fractal has a name: the Sierpinski Triangle.

A **fractal** is a geometric shape that can be (infinitely) broken down into similar parts. This means that it is made up of many parts that are just smaller versions of the whole thing. And these smaller parts are then made up of even smaller versions of the whole thing. And this goes on infinitely (well, basically). Of course, our eyes can't see this going on infinitely because things get too small.

Randomness and probability

In the chaos game, random vertices of an equilateral were repeatedly selected. What does *random* mean exactly? We have seen that there is sometimes order in chaos if you look long and close enough. That

being said, there are things that seemingly happen in a haphazard manner, have no predictable pattern, and are not predetermined. Such things are called random. Think of selecting one of the three vertices. We have no way of predicting which vertex will be selected. Any one could be selected at each iteration. Think of rolling a die. There is no way to predict which number will be rolled next (other than to say that it will be between 1 and 6 inclusive).

Even with random things, human beings have come up with a way of measuring the randomness, and that way is called probability. Probability is a way of expressing the knowledge that something will happen. If something is definitely going to happen (i.e., it's absolute certainty), it is said to have a probability of 1. If something is definitely **not** going to happen (i.e., it's an impossibility), it has a probability of 0. Of course, there are an infinite amount of values in between 0 and 1. That is, the probability that something could happen can be stated as a value between 0 and 1.

Formally, probability is defined as the ratio of the number of ways of achieving success to the total number of possible outcomes. For example, consider the flipping of a coin. There are two possibilities when flipping a coin: either heads or tails. So there are two possible outcomes. The probability of landing on heads when flipping a coin is then 1/2. Why? The only way to land on heads is, well, to flip heads. That is, there is only one way to achieving success. Since there are two possible outcomes when flipping a coin, the probability of landing on heads is therefore 1/2 (one way of achieving success out of two possible outcomes). Similarly, the probability of landing on tails when flipping a coin is also 1/2.

What is the probability of picking the top vertex in the chaos game? There's only one way to do so out of three possible vertices. Therefore, the probability is 1/3.

Let's go back to rolling a typical (six-sided) die. What is the probability that a three is rolled? Rolling a three represents the only way to achieve success. There are six possible rolls of the die. Therefore, the probability of rolling a three is 1/6.

What is the probability that an even number is rolled? Well, how many ways are there of achieving success? That is, how many ways can an even number be rolled? Three (rolling a two, four, or six). The possible outcomes are, of course, rolling a one through six. So the probability of rolling an even number is 3/6 = 1/2.

What is the probability that a number less than six is rolled? There are five ways to roll a number less than six. There are six possible rolls of a die. The probability is therefore 5/6.

What is the probability of rolling two die and getting a total of 7? This one is a bit tricky! We'll discuss this later (but if you want to know, it's 1/6).

Activity 4: Heads and tails...sort of

For this activity, we will repeatedly flip two coins simultaneously and record the results. The two coins can either both be heads, both be tails, or they can be different (i.e., one is heads and one is tails). So there are three possible outcomes to this game. The probability of both coins landing on heads is therefore 1/3, both landing on tails is also 1/3, and one landing on heads and the other on tails is also 1/3.

To make this interesting, let's make it a game. If both coins land on heads, all of the students seated in the left half of the class (on the left side from the prof's perspective) get a point. If both coins land on tails, all of the students seated in the right side of the class get a point. If one coin lands on heads and the other lands on tails, the prof gets a point. The group (or individual) with the most points at the end of some number of flips (say, 15) wins.

Record the results below:

Left	**Right**	**Prof**

Over time, you will find that the students lose more often. It appears that the heads and tails combination happens significantly more often than initially thought. In fact, the previously calculated probabilities (each 1/3) seem incorrect.

The purpose of this activity is to show that perceived probabilities are often different from actual probabilities (i.e., humans are not always so good at estimating probabilities). Think of it like this: there is only one way of both coins landing on heads. There is only one way of both coins landing on tails. Contrary to what was previously assumed, there are two ways of ending up with one coin landing on heads and the other landing on tails: the first coin can be heads and the second coin can be tails, or the first coin can be tails and the second coin can be heads! In fact, there are a total of four possible outcomes. The flipping of two coins can then be shown in the following table:

Coin 1	Coin 2
heads	heads
heads	tails
tails	heads
tails	tails

The probability of both coins landing on heads is 1/4 (one possible way of achieving success out of four possible outcomes as show above). The probability of both coins landing on tails is also 1/4. But the probability of one coin landing on heads and the other landing on tails is actually 2/4 = 1/2! The prof wins 50% of the time! Another way of saying this is that the prof wins twice as much as either group of students! Ka-ching!

When you realize how rigged this simple *game* is, you can begin to think about how even more rigged other games (such as slot machines, the lottery, etc) are...

Try to calculate the probability of flipping three coins and getting three heads. Or three tails. Or one heads and two tails. Or one tails and two heads. To assist you, list all of the possible combinations of flipping three coins in the table below:

Coin 1	Coin 2	Coin 3

Probability of ending up with three heads:

Probability of ending up with three tails:

Probability of ending up with one heads and two tails:

Probability of ending up with one tails and two heads:

Random number generators

In the last RPi activity (titled *My Binary Addiction...Reloaded*), you may have noticed the use of a library that allowed the generation of random numbers. In the activity, it was used to generate random bits (0 or 1) for each of the two 8-bit numbers (in order to generate two random 8-bit numbers). The library was called `random` and was imported as follows:

```
from random import randint
```

The function `randint` was then used to generate the random numbers. As mentioned earlier in this lesson, a truly random event has no predictable pattern. Unfortunately, there is no easy way for that to happen using a computer, even for a task as simple as selecting random numbers within a specific range. As a solution, computers typically use a pseudo-random process to create random numbers in a range.

Definition: *A **pseudo-random process** is one that appears to be random but is technically deterministic in nature. This means that the process looks completely random, and yet its pattern is predictable and can be reproduced exactly.*

When tasked with creating a random number (or group of numbers), computers typically use a pseudo-random process to produce that number. This means that, given enough generated numbers, one can observe that the created numbers follow a certain sequence. For most purposes, however, the amount of numbers one would require to observe that the pattern is predictable is too large (so the numbers seem random).

For most cases, pseudo-random numbers are adequate and even beneficial. How? For instance, often researchers replicate experiments in order to confirm (or refute) experiments performed by peers. For a

scientist to make any claim about any experiments carried out, enough information for someone else to reproduce those same results with an experiment of their own must be provided. When a random number (or list of random numbers) is used in an experiment, there would be no way of replicating those exact numbers (and therefore obtaining the same results as in the original experiment). However, since computers produce pseudo-random numbers (which can be replicated exactly), it allows other researchers to carry out their own versions of the original experiment and crosscheck to see if the results are identical.

In order to actually generate repeatable (even predictable) patterns of *random* numbers, most pseudo-random number generators can be configured or initialized with a seed. A **seed** is just a number that is used to initialize a pseudo-random number generator. Most generators use the current time (to the second) as the seed. Clearly, this means that generating a sequence of random numbers will be different each time (since time moves forward continuously). We can, however, specify a seed of our own so that the pseudo-random number generator will always provide the same sequence of numbers!

Activity 5: Seeded pseudo-random numbers

Let's create a simple program generates 100 pseudo-random numbers (in the range 0-99) twice:

```
from random import randint

for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
print
for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
```

Note the use of `"\t"` (which prints a *tab* and is typically eight spaces in width). Characters that begin with a backslash (\) are known as **escape characters**. Escape characters typically allow the representation of unprintable characters (such as a tab, newline, carriage return, etc). Most general purpose programming languages (including Python) support these unprintable characters. Here are some common ones:

`\n`	Linefeed (like pressing Enter)
`\t`	Horizontal tab

Note that there are many others, some of which are even programming language specific.

The program above first displays 100 random integers in the range 0-99, and formats them such that ten integers are displayed on each line (i.e., ten rows of ten columns). They are aligned at the columns via a horizontal tab that follows each integer. Note the comma at the end of the print statement. This instructs Python to omit a typically default linefeed at the end of the print statement. Once ten integers have been displayed on a single line (i.e., `i % 10 == 0`), a blank line is added (via the solitary print statement). The program then displays another 100 random integers in the same manner. Note the variety (and randomness) of the generated integers, although generating 200 integers in the range 0-99 will undoubtedly produce duplicates.

In the above example, the pseudo-random number generator was seeded with the current time (by default since no numeric seed was provided). Let's modify the algorithm and use an identical (but specified) seed value for each 100 random integers:

```
from random import randint, seed

seed(123456)
for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
print
seed(123456)
for i in range(1, 101):
    print "{}\t".format(randint(0, 99)),
    if (i % 10 == 0):
        print
```

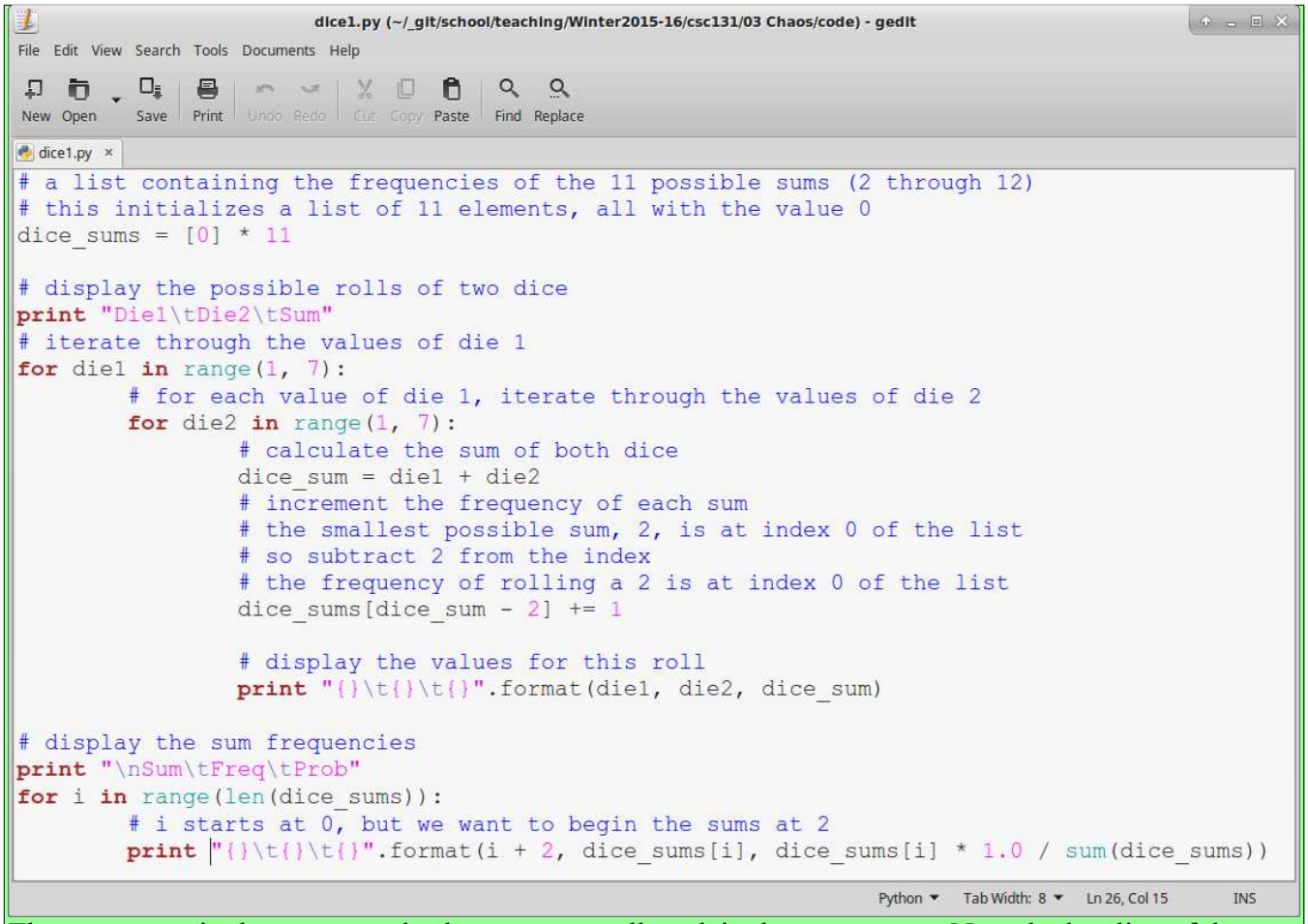
In this case, the numeric value 123456 is used as the seed to generate both groups of 100 integers. Technically, the seed could be any value (e.g., 0, some variable containing a value, even the mathematical constant pi). In fact, we could have randomly generated a seed!

```
seed(randint(0, 65535))
```

Now that you have seen how to use Python to generate pseudo-random integers, let's try to write programs that implement some of the activities shown earlier in this lesson.

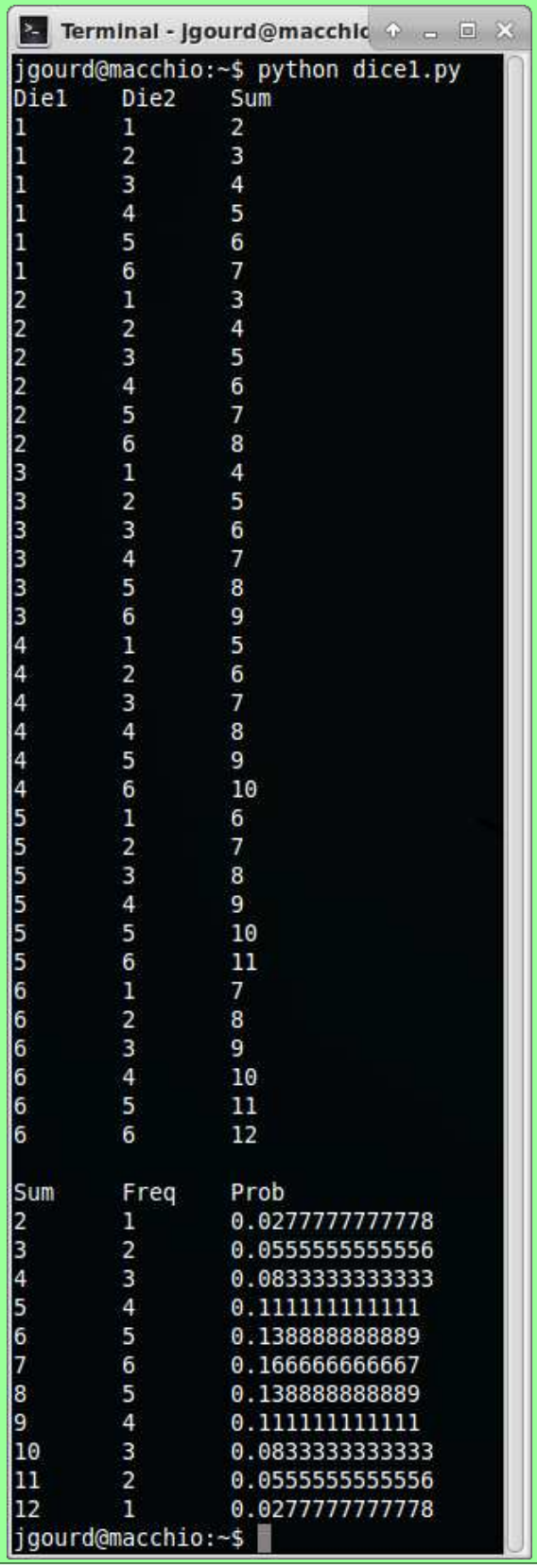
Activity 6: Rolling two dice

Earlier we discussed rolling a single die and the associated probabilities of each possible roll. Then, we asked what the probability of rolling two dice and getting a total of 7. Let's try to write a Python program that iterates through all of the possibilities when rolling two dice:



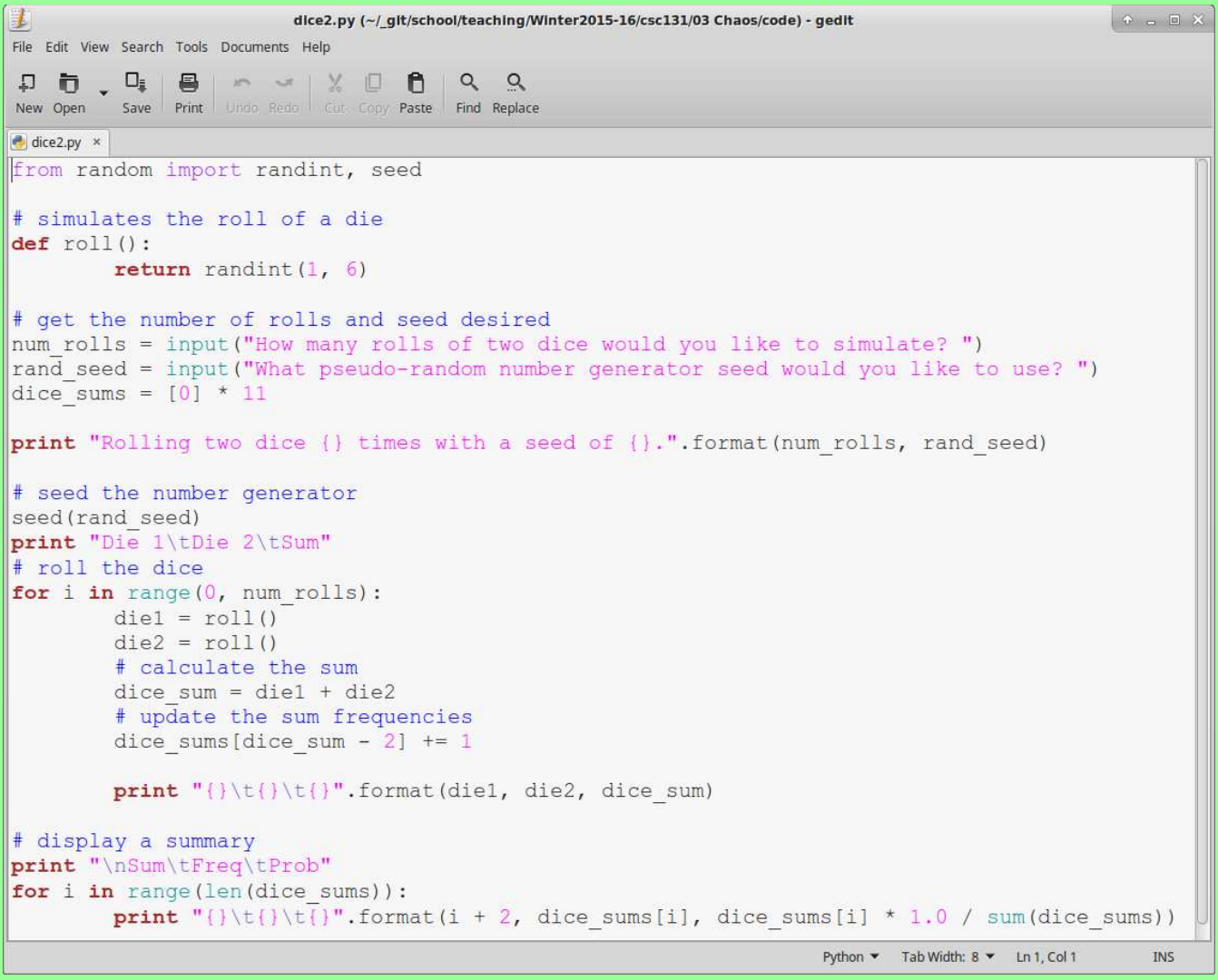
The comments in the source code above pretty well explain the statements. Note the last line of the program; specifically, the third parameter that replaces the formatting braces: `dice_sums[i] * 1.0 / sum(dice_sums)`. This expression takes the i-th sum frequency and converts it to a floating point number by multiplying it by 1.0. This product is then divided by the total number of sum frequencies (the function `sum()` calculates the sum of a list – in this case, the list `dice_sums`). The conversion of one of the terms to floating point is necessary to ultimately produce a floating point division.

And here is the output of the code:



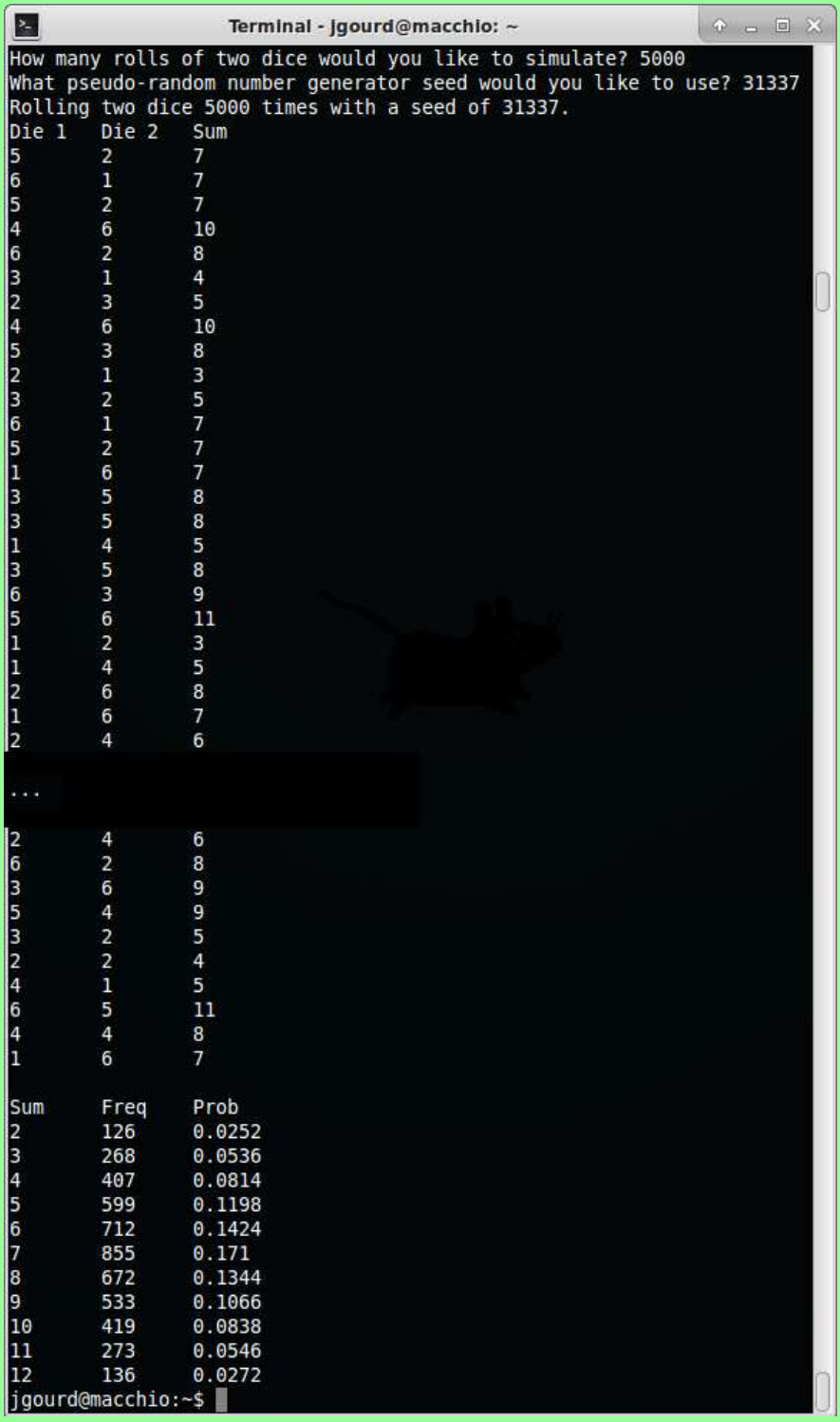
The number of total sums possible is 36. We can calculate this by taking the sum of the frequencies (1 + 2 + 3 + 4 + 5 + 6 + 5 + 4 + 3 + 2 + 1 = 36). Since there are six rolls that sum to 7, then the probability of rolling two dice and getting a 7 is 6/36 = 1/6. In fact, the probability of rolling two dice and getting a 7 is higher than anything else!

Now let's try to roll two dice many times and see what results we end up with. Perhaps the frequencies of the sums will match the frequencies shown above! Here's the code:



Again, the comments should be adequate to explain the source code. Much of the code is similar to the last program.

And here is the output (with 5,000 rolls):



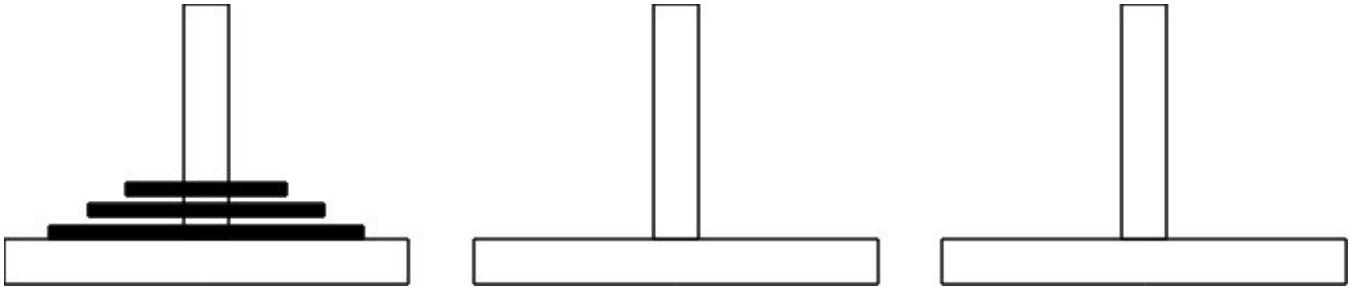
Note the probabilities. Although they are not exactly the same as calculated before, they are very similar! More accurate results would be possible with a much higher number of rolls (try one million!).

The Towers of Hanoi

Although a lot of problems exist, it is often fun (and interesting) to study games to see if we can glean anything of value from them beyond just entertainment. In some cases, we actually learn useful tactics that can help us solve *real* problems later on. The Towers of Hanoi is an old, simple game (in principle). You are presented with three pegs (or towers) on which you can move various discs around. Initially, the discs rest on a single tower (the largest is on the bottom and the smallest is on the top). The objective is to move the discs from this *source* tower to some *destination* tower. But there are rules:

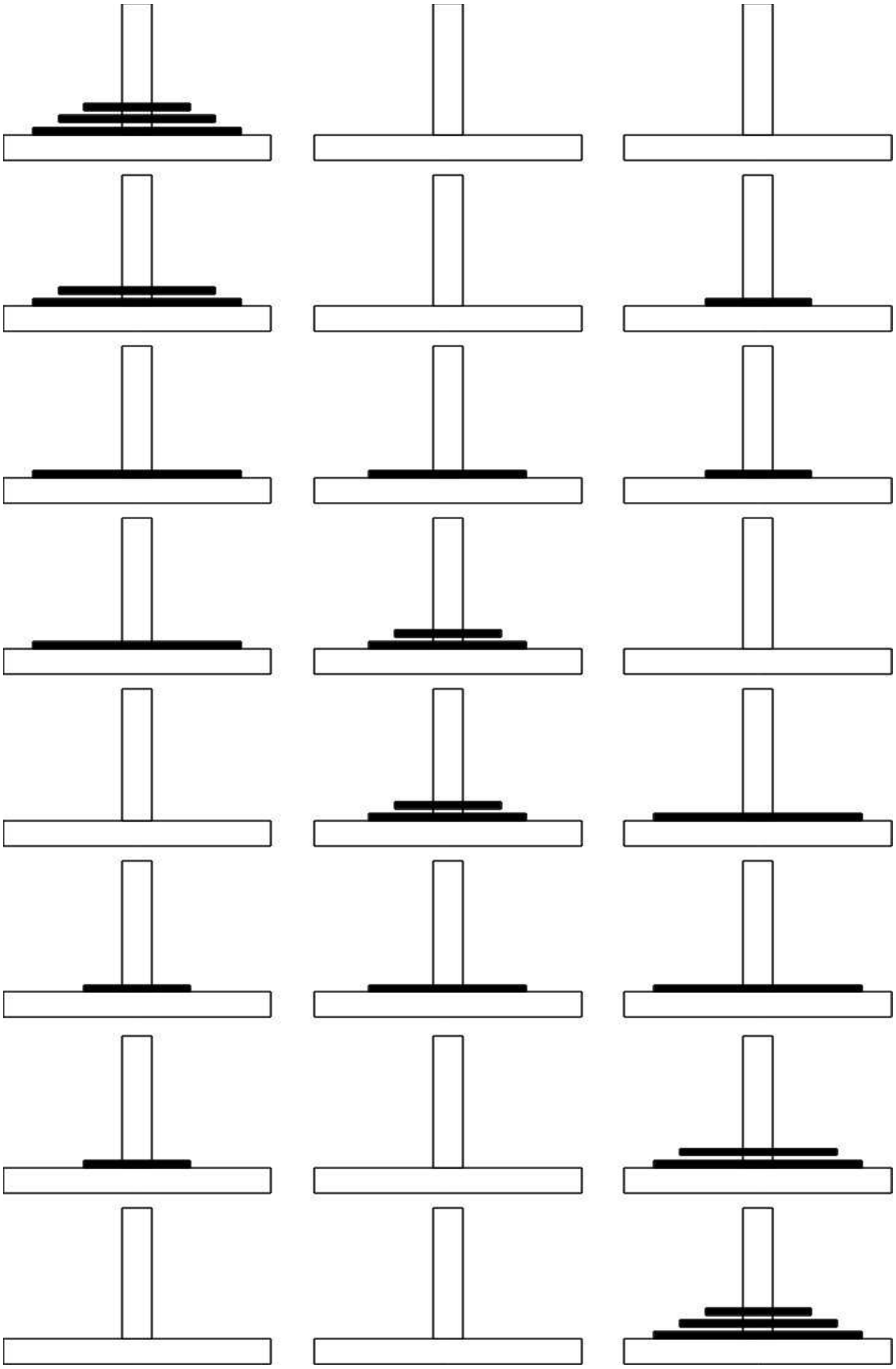
1. Only a single disc can be moved at a time;
2. No larger disc may ever be placed on top of a smaller disc; and
3. Moving a single disc constitutes one move.

The objective, of course, is to move all of the discs from a *source* tower to a *destination* tower in the minimum number of moves. Here is an example of what the start of the game looks like (with three discs):



The game is interesting because it presents us with an optimization problem. That is, the goal is not just to find a solution (i.e., to successfully move the discs from the source tower on the left to the destination tower on the right), but to find one that is optimal (i.e., that results in some minimum number of moves). It is quite easy to eventually find a solution in, say, 100 moves. In fact, we can probably make mostly random moves and get there within 100.

The minimum number of moves required to solve the Towers of Hanoi with three discs turns out to be seven! In fact, here are the moves:



Trying our hand at solving the puzzle with one, two, three discs, and so on, may lead to some idea of where to place the first disc. We also notice a pattern; however, it is not so straightforward. In fact, attempting to design an algorithm to solve the puzzle would most likely be difficult at this point. It would undoubtedly involve repetition, and breaking down the problem so that it can be reasoned about and an algorithm designed is not as simple as it may initially appear.

The simplest case of the Towers of Hanoi is one with a single disc. The minimum number of moves required in this case is, trivially, one (move the disc from the source tower to the destination tower). We can create a table that, given an initial number of discs, provides the minimum number of moves required for a Towers of Hanoi problem with that many discs:

Number of discs	Minimum number of moves
1	1
2	3
3	7
4	15
5	31
6	63

In the general case of *n* discs, how many moves would it minimally take? Do you see a pattern? A quick look indicates that the minimum number of moves required appears to double each time a disc is added. For example, one disc requires one move. Two discs requires three moves (a bit more than double the number of moves required for a single disc). Three discs requires seven moves (a bit more than double the number of moves required for two discs). And so on.

It appears that the minimum number of moves required can be represented as a power of two of the number of discs. In the case of a single disc, we have: $2^1 = 2$; and $2 - 1 = 1$. In this case, that's $2^1 - 1$ moves (minimally). In the case of two discs, we have: $2^2 = 4$; and $4 - 1 = 3$. That's $2^2 - 1$ moves. And in the case of three discs, we have: $2^3 = 8$; and $8 - 1 = 7$. That's $2^3 - 1$ moves. Generalizing this for *n* discs, we simply have $2^n - 1$ moves (minimally)! Here's the updated table:

Number of discs	Minimum number of moves
1	1
2	3
3	7
4	15
5	31
6	63
n	$2^n - 1$

Let's suppose that it takes 1 second for you to make a single move. How long would it take to complete the puzzle with six discs? Since it takes 63 moves to complete the puzzle with six discs, and each move

takes 1 second, then it would take slightly over one minute to complete the puzzle. Since the number of moves effectively doubles each time a disc is added to the problem, so does the time it takes to solve the puzzle! For example, it would take approximately two minutes to solve a puzzle with seven discs, approximately 16 minutes for a puzzle with ten discs, and approximately 11 days for a puzzle with 20 discs!

Suppose that your computer could make 500 million moves per second (which is probably about right!). A quick calculation shows us that the Towers of Hanoi with 30 discs would take a little over two seconds. Again, it doubles with every additional disc. For example, it would take your computer approximately four seconds for a puzzle with 31 discs, approximately one minute for puzzle with 35 discs, and approximately one hour for a puzzle with 41 discs. And it would take almost one month for a puzzle with 50 discs! One month for a fast computer that can make 500 million moves per second!

It is said that a group of monks are working to solve the Towers of Hanoi with 64 golden disks. They believe that, once solved, it will be the end of the world. They move the discs by hand. Think of the size of the discs to be able to stack 64 of them (golden ones, mind you) on a single tower! They must require pulleys, ropes, many monks for each move, and so on. Suppose that they can make one move every ten minutes (which is quite fast, considering). It would take them over 350 billion millennia (that's over 350 billion thousand years) to solve the puzzle in the minimum ~18.4 quintillion moves! I'm not worried...

Breaking problems down

Consider a simple question: Let's say that there are ten students in the class. Suppose that some activity requires pairing students in groups of two. How many unique groups could be generated? That is, how many ways could the class be split up into groups of two? Although not particularly difficult, the answer is not obvious at first. A strategy may be to simplify the problem into a trivial case, and build up from there. We could do this by first considering a class of two students. How many groups of two could be formed? Clearly, only one. There is only one way to group two students in groups of two. Let's add a third student. In fact, let's consider the three students, S_1, S_2, and S_3. How many ways could groups of two be formed from these three students? Let's try to enumerate them all:

1. S_1 and S_2;
2. S_1 and S_3; and
3. S_2 and S_3.

So there are three ways to form groups of two from three total students. What about four total students, S_1 through S_4?

1. S_1 and S_2;
2. S_1 and S_3;
3. S_1 and S_4;
4. S_2 and S_3;
5. S_2 and S_4; and
6. S_3 and S_4.

There are six ways to form groups of two from four total students. One more: what about five total student, S_1 through S_5?

1. S_1 and S_2;
2. S_1 and S_3;
3. S_1 and S_4;
4. S_1 and S_5;

5. S_2 and S_3;
6. S_2 and S_4;
7. S_2 and S_5;
8. S_3 and S_4;
9. S_3 and S_5; and
10. S_4 and S_5.

There are ten ways to form groups of two from five total students. How does this scale to more students? Take a look at this table (note that there are exactly zero ways to form groups of two from a single student!):

Students	Groups
1	0
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	?

Can you guess the number of groups of two that can be formed with ten students? Can you see a pattern that gives the number of groups from some number of students? Take a closer look at a subset of the table above:

Students	Groups
1	0
2	1
3	3

For the case of three students, it seems that we can calculate the number of groups of two that can be formed as the sum of the number of groups of two that can be formed with two students plus those two students (i.e., 1 + 2 = 3). Let's see if this continues to work out with four students:

Students	Groups
1	0
2	1
3	3

4	6

To calculate how many groups of two can be formed with four students, we can sum the number of students in the row above (3) with the number of groups that can be formed with that many students (3): (3 + 3 = 6).

And one more:

Students	Groups
1	0
2	1
3	3
4	6
5	10

The number of groups of two that can be formed with five students is the number of groups of two that can be formed with four students (6) plus those four students (4): (6 + 4 = 10). And now, we can calculate the number of groups of two that can be formed with 10 students:

Students	Groups
1	0
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	45

How can we generalize this for any number of students (say, *n*)? It turns out that we can use a recurrence relation to describe this behavior.

Definition: *A **recurrence relation** is an equation that recursively defines a sequence. That is, each term in the sequence is defined as a function of the preceding terms in some way.*

For example, in the table above the number of groups of two that can be formed with ten students is a function of the number of groups of two that can be formed with nine students. And that itself is a function of the number of groups of two that can be formed with eight students. And this goes on. Ultimately, however, we can establish some base (or trivial) case that is immediately answerable. For

example, we can say (without needing to think about it too much) that no groups of two can be formed with one student. In fact, this is the simplest case for this problem. It is the base (or trivial) case.

Recurrence relations must ensure that, at some point, the base case is achievable. That is, the problem must be repeatedly broken down into smaller and smaller versions of itself, eventually reaching the base case. The solution to a specific term in the sequence is then built back up, from the base case! Algorithms that repeatedly break something down until a base case is reached and build a solution back up are known as **divide and conquer** algorithms.

For the groups of students problem stated above, we can define a function, $G(n)$, that calculates the number of groups of two that can be formed from n total students. We could express this function as follows:

$$G(n)=\begin{cases}0 & \text{, if } n=1\\ (n-1)+G(n-1) & \text{, otherwise}\end{cases}$$

We read this as follows: the number of groups of two that can be formed from n students is:

- 0, if the number of students, n, is 1; or
- n minus 1 plus the number of groups of two that can be formed from n minus 1 students, otherwise.

Although fully understanding this at this point is not necessary, note how the second part of the function is dependent on the function itself. That is, the *otherwise* part breaks the problem down a bit into a smaller version of itself via $G(n-1)$. The interesting thing is that the problem can be broken down enough times to ensure that the base case is eventually reached! If n is some positive number greater than 0, the recurrence relation will break down the problem until n is 1, at which point the base case is reached (and everything stops).

Formally, the equation above is known as a recurrence relation because it is defined in terms of itself. It also has two separate parts, and the result depends on the input value. In mathematics, we refer to this type of function as a piecewise function. This particular function has two pieces:

- The first results in 0, but only if the input value, n, is 1; and
- The second results in a broken down version of itself, otherwise (i.e., for values of n that are not 1).

Recursion

In computer science, recursion is usually understood to be the idea of a subprogram repeatedly calling itself. Of course, at some point this repeated calling has to stop (otherwise, it would be an infinite loop). In a previous lesson, we envisioned recursion as a spiral of sorts. Each time a subprogram calls itself, we descend down a level of the spiral until we eventually reach the bottom (some base or trivial case). At that point, execution begins to *unwind* as the subprogram calls complete and we retrace our path back up through the various levels until finally arriving at the *top* level where execution began. This is when the solution is built back up. Generally, however, recursion is just another name for recurrence relation.

Definition: ***Recursion*** *is the process of breaking down a problem into smaller and smaller versions of itself until a base or trivial case is reached. Recursion must have two parts: (1) a base or trivial case that provides an immediate answer to some specified input; and (2) a recursive step that breaks the problem down into a smaller version of itself.*

Recursion is the programming equivalent of mathematical induction (which is just defining something in terms of itself). To illustrate this more clearly, let's take a look at a simple algorithm that computes a base to some power (i.e., x^y):

```
# compute 2^10
x = 2
y = 10
pow = 1

for i in range(0, y):
     pow *= x

print "{}^{} = {}".format(x, y, pow)
```

This example computes 2^{10} (1024). It basically multiplies one by the base (x) y times. When the base is two and the exponent is ten, it multiples one by two, ten times (i.e., 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 *2 * 2).

To convert this iterative algorithm to a recursive one, we must first define the recurrence relation that solves the problem. First, the base or trivial case. For exponents, that's simple: $x^0 = 1$. That is, anything raised to the power zero is always one. The recursive step takes more thought. A hint is to observe how powers can be broken down. For example: $2^2 = 2 * 2$. Extending this: $2^3 = 2 * 2 * 2$. This can be rewritten as $2^3 = 2 * 2^2$. We can extend this further: $2^4 = 2 * 2^3$. Notice how the exponents can be repeatedly broken down into smaller exponents. The trick is to see if the base case is eventually reached.

Let's take a look at the first example again: $2^2 = 2 * 2$. Technically, $2^2 = 2 * 2^1$. And $2^1 = 2 * 2^0$! So the base case can eventually be reached. We can now formally define a recurrence relation for some function `Pow(x, y)` as follows:

$$Pow(x,y)=\begin{cases}1 & \text{, if } y=0\\ x*Pow(x,y-1) & \text{, otherwise}\end{cases}$$

Let's see if it works with an example: 2^5 (which is 32). In the recurrence relation above, we would call the function with Pow(2, 5). Since y is not zero, the second (recursive) case is applied resulting in 2 * Pow(2, 4). To calculate Pow(2, 4), the second case is applied again resulting in 2 * Pow(2, 3). And to calculate Pow(2, 3), we apply the second case another time which results in 2 * Pow(2, 2). We have to apply the second case a few more times: first with Pow(2, 2) which results in 2 * Pow(2, 1), and lastly with Pow(2, 1) which results in 2 * Pow(2, 0). At this point, the first (base) case is applied (since y is 0) resulting in 1. We can now build it all back up: 1 * 2 * 2 * 2 * 2 * 2 = 32.

Perhaps this is best illustrated as follows; first with the recursive calls:

```
Pow(2, 5)
        2 * Pow(2, 4)
                2 * Pow(2, 3)
                        2 * Pow(2, 2)
                                2 * Pow(2, 1)
                                        2 * Pow(2, 0)
```

Now let's build the result back up:

~~Pow(2, 5)~~ 32
2 * ~~Pow(2, 4)~~ 16
2 * ~~Pow(2, 3)~~ 8
2 * ~~Pow(2, 2)~~ 4
2 * ~~Pow(2, 1)~~ 2
2 * ~~Pow(2, 0)~~ 1

Notice how, at each step in the building back up, the result of the recursive calls combine to form simple arithmetic problems (e.g., 2 * Pow(2, 0) becomes 2 * 1 which equals 2). The arithmetic results form the answer to a previous recursive call (which is then replaced to form another simple arithmetic problem).

Here's another interesting mathematical function: the factorial. Let's first look at its recursive definition:

$$Fact(n) = \begin{cases} 1 & \text{, if } n=0 \\ n * Fact(n-1) & \text{, otherwise} \end{cases}$$

The factorial function repeatedly multiplies some integer by all the integers below it (up to 1). For example, five factorial (referred to as 5!) = 5 * 4 * 3 * 2 * 1 = 120. To see how this works, let's take a look at some examples:

5! = 5 * 4 * 3 * 2 * 1 = 120
4! = 4 * 3 * 2 * 1 = 24
3! = 3 * 2 * 1 = 6
2! = 2 * 1 = 2
1! = 1

Note how 4! is embedded within 5!:

5! = 5 * 4 * 3 * 2 * 1

And how 3! is embedded within 4!:

4! = 4 * 3 * 2 * 1

And so on. In fact, we could say that 5! = 5 * 4!, and 4! = 4 * 3!, and 3! = 3 * 2!, and so on. This clearly breaks the problem down into smaller and smaller versions of itself. So what is the base case? Perhaps it's just that 1! = 1. Although correct, mathematicians actually prefer 0! = 1 (as illustrated in the recurrence relation above). So the breaking down occurs as follows:

5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1

And now we can implement a recursive factorial function as follows:

```
def fact(n):
    if (n == 0):
        return 1
    else:
        return n * fact(n - 1)
```

And to compute 5!:

```
print "5! = {}".format(fact(5))
```

Let's see how the recursion works as in the previous Pow example; first with the recursive calls:

```
fact(5)
    5 * fact(4)
        4 * fact(3)
            3 * fact(2)
                2 * fact(1)
                    1 * fact(0)
```

Next with the building back up:

```
~~fact(5)~~  120
    5 * ~~fact(4)~~  24
        4 * ~~fact(3)~~  6
            3 * ~~fact(2)~~  2
                2 * ~~fact(1)~~  1
                    1 * ~~fact(0)~~  1
```

Did you know?

Although not always simple, any iterative algorithm can be converted to a recursive one, and vice versa!

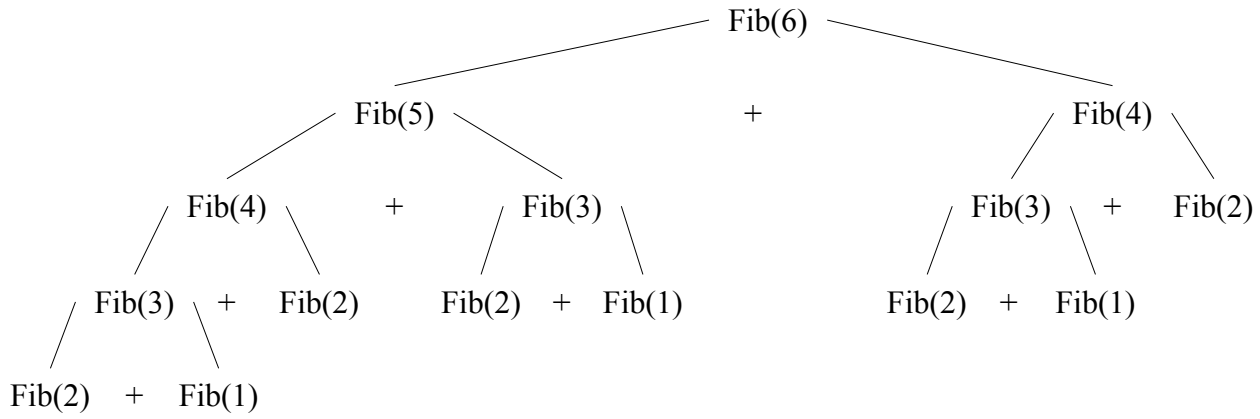
Try to implement an iterative factorial algorithm (in Python) in the space below:

Another interesting mathematical function is one that generates the Fibonacci sequence. Although it has been featured in a puzzle, no actual function was provided. Let's take a look at the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

A term in the sequence can be calculated as the sum of the two previous terms. For example, the seventh term (8) is the sum of the fifth and sixth terms (3 and 5). Here's a recurrence relation for the Fibonacci sequence:

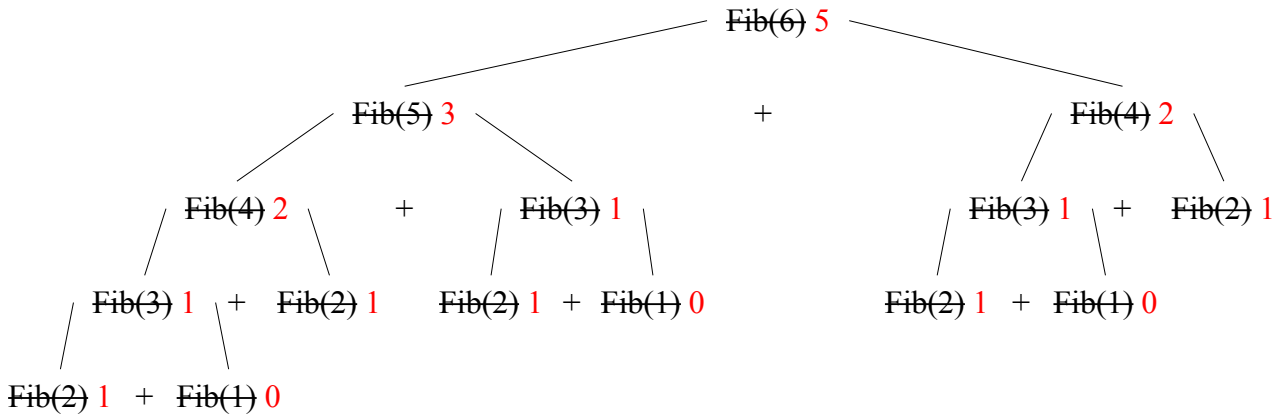
$$Fib(n) = \begin{cases} 0 & \text{, if } n = 1 \\ 1 & \text{, if } n = 2 \\ Fib(n-1) + Fib(n-2) & \text{, otherwise} \end{cases}$$

Note that this recurrence relation has two base cases! This is necessary since the recursion requires the sum of the *two* previous terms. The first term in the sequence, Fib(1), is 0 (the first base case). The second term in the sequence, Fib(2), is 1 (the second base case). The remaining terms are calculated recursively as the sum of the previous two terms, Fib(n – 1) + Fib(n – 2). Here's how the recursion breaks down for Fib(6) (which equals 5). Since there are multiple branches of recursion (i.e., two parts to each recursive call), we'll use a different method to show the recursion:



At the top, Fib(6) breaks down to Fib(5) + Fib(4). The figure then takes on an upside-down tree-like structure. Note how, if flipped upside-down, the top would form a root. The root then splits into two branches, each of which split into two more branches, and so on. The values at the bottom of the figure form what are called leaves. These represent the base cases, all either Fib(1) or Fib(2). In a future lesson, we will discuss trees more formally.

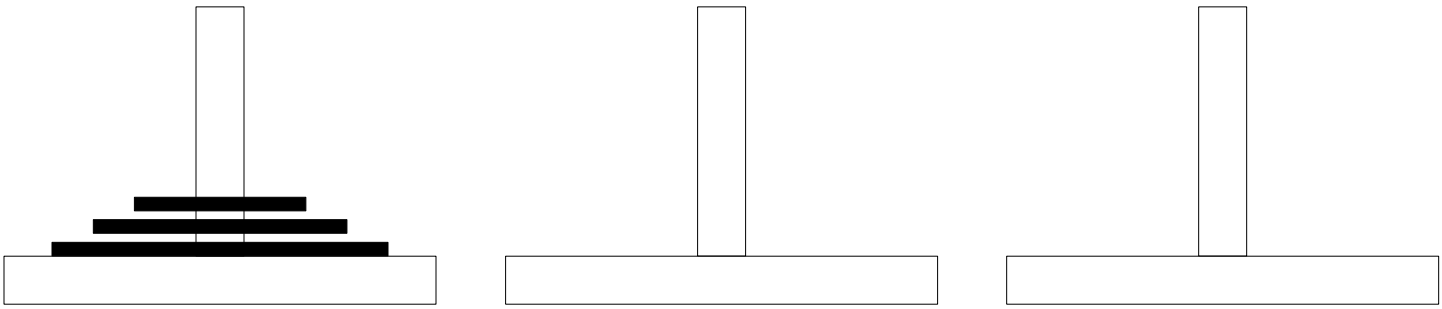
We can build the result back up as follows:



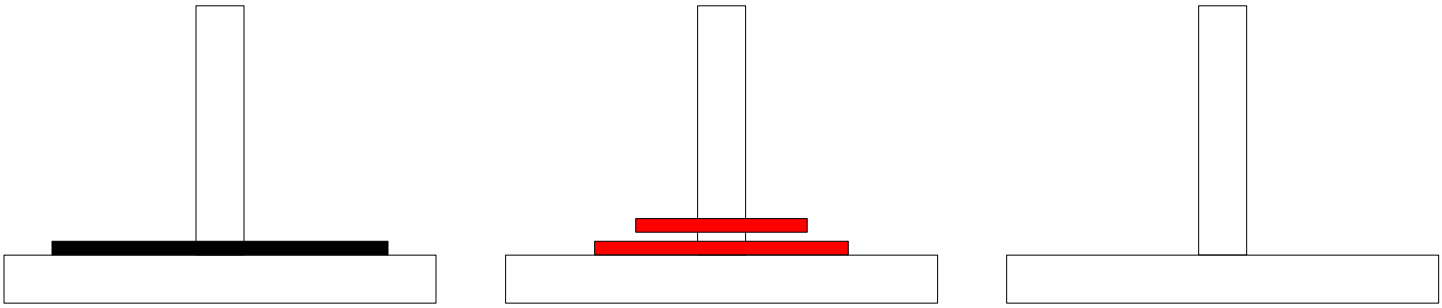
The result (at the top) is, as expected, 5.

The Towers of Hanoi...Reloaded

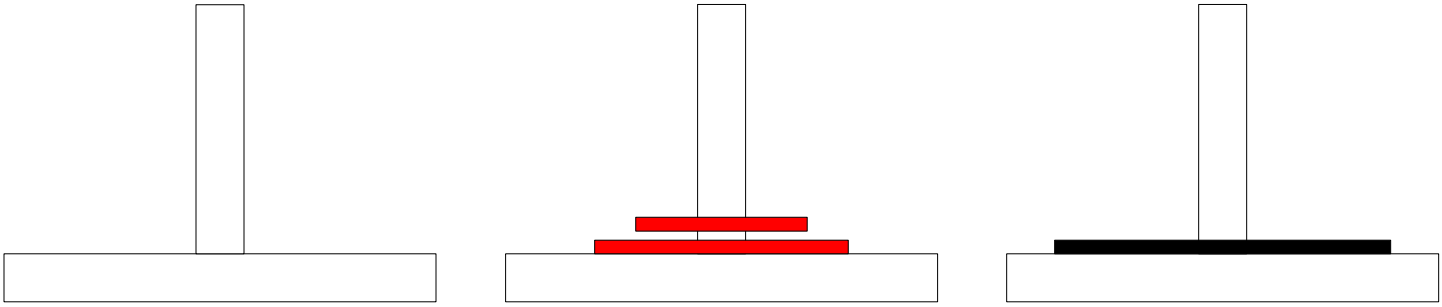
As noted earlier, crafting an algorithm for this puzzle is not as simple as it sounds. In fact, it is quite difficult to design an iterative algorithm for it. Let's see if we can use recursion to our advantage. Consider the puzzle with three discs:



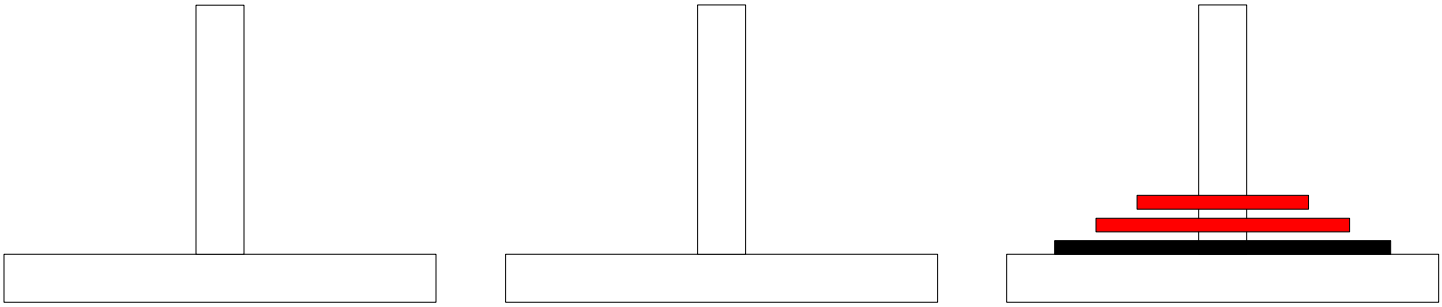
Suppose that the destination tower is the one on the right. The middle tower will be used as a spare. In order to move all three discs from the source tower to the destination tower, we can think of first needing to move two discs (the top two, in fact) from the source tower to the spare tower:



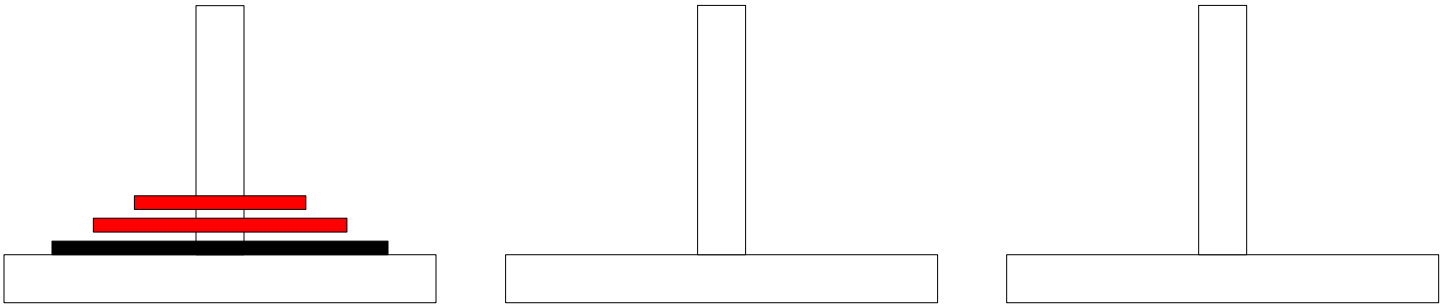
Clearly this is not possible since we cannot move two discs simultaneously; however, let's continue with this *train of thought* for a moment. Supposing that we have successfully done this, then we would need to move the single disc left on the source tower to the destination tower:



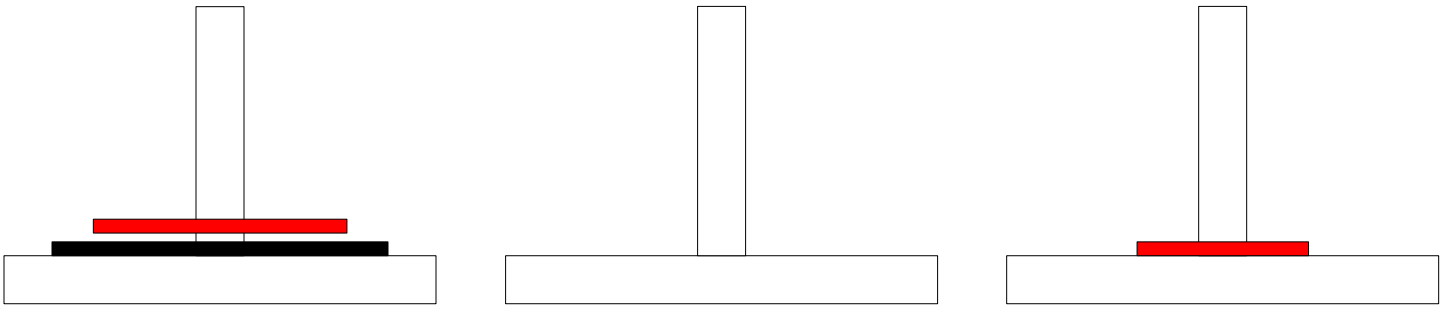
And all that would be left to do is to move the two discs that are on the spare tower over to the destination tower:



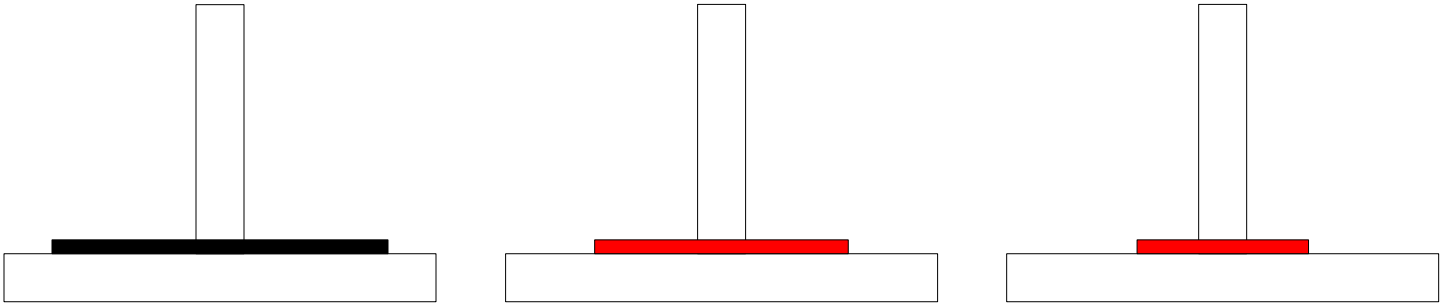
Puzzle solved! However, as mentioned we cannot move two discs simultaneously. However, let's go back to the original state:



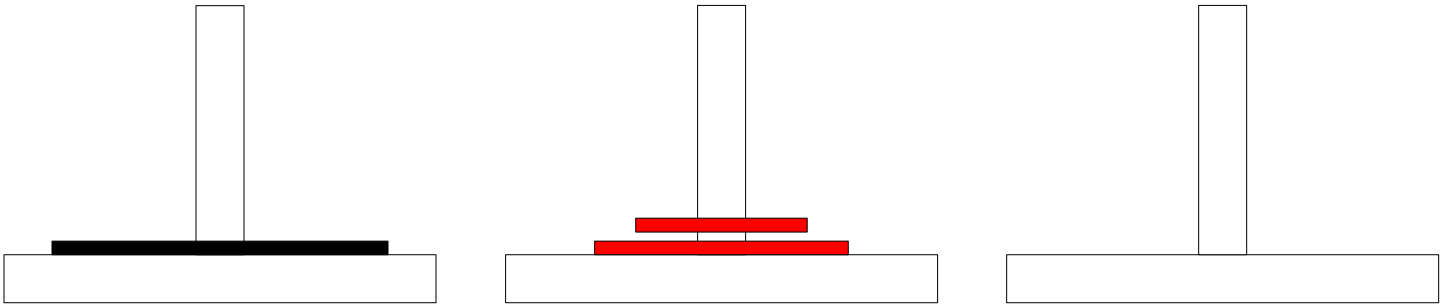
In order to move the two top discs from the source tower to the spare tower, we first need to move the single top disc from the source tower to the destination tower:



We can then move the second disc from the source tower to the spare tower:



And we can finally move the small disc from the destination tower to the spare tower:



In a sense, the three disc puzzle (i.e., moving three discs from the source tower to the destination tower) is a two-disc puzzle (moving the two top discs from the source tower to the spare tower), followed by a one-disc puzzle (moving the largest disc from the source tower to the destination tower), and finished with a two-disc puzzle (moving the two discs left on the spare tower to the destination tower).

But of course, the two-disc puzzles are simply three one-disc puzzles! And a one-disc puzzle is simple: just move the disc from one tower to another. Moving one disc is, in fact, the base case of the Towers of Hanoi! Moving more than one disc can be broken down as a sequence of three smaller puzzles as follows:

1. Move all but the bottom disc from the source tower to the spare tower;
2. Move the largest disc from the source tower to the destination tower; and
3. Move the discs on the spare tower to the destination tower.

Or in general, given *n* discs:

1. Move *n* – 1 discs from source to spare;
2. Move 1 disc from source to destination; and
3. Move *n* – 1 discs from spare to destination.

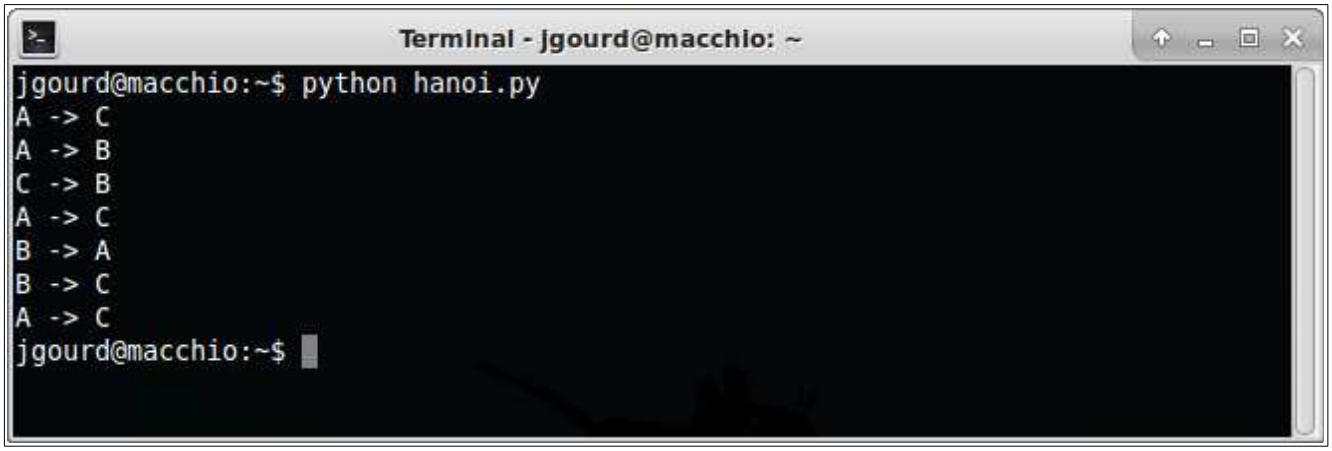
We can design a recursive algorithm in Python as follows:

```
def hanoi(n, src, dst, spr):
    if (n == 1):
        print "{} -> {}".format(src, dst)
    else:
        hanoi(n - 1, src, spr, dst)
        hanoi(1, src, dst, spr)
        hanoi(n - 1, spr, dst, src)
```

Note the parameters in the function `hanoi`. The variable *n* specifies the number of discs. The variables *src*, *dst*, and *spr* refer to the three towers. We can execute the algorithm and call the `hanoi` function (with three discs from a source tower A to a destination tower C using a spare tower B) as follows:

```
hanoi(3, "A", "C", "B")
```

Here is the output:



The recursive function does require a little bit of explanation. Let's take a look at the initial call to the function again:

```
hanoi(3, "A", "C", "B")
```

When this call occurs, the actual parameters, 3, “A”, “C”, and “B”, are passed in (or mapped) to the formal parameters, *n*, *src*, *dst*, and *spr*. To be clear, the source tower is A, and the destination tower is C. The first part of the recursive function, representing the base case when *n* = 1, is clear: simply move the single disc from *src* to *dst*. This is accomplished by displaying the move to the console:

```
print "{} -> {}".format(src, dst)
```

If $n > 1$, the recursive step is applied:

```
hanoi(n - 1, src, spr, dst)
hanoi(1, src, dst, spr)
hanoi(n - 1, spr, dst, src)
```

That is, to move *n* discs from *src* to *dst* using *spr*, we must first move $n - 1$ discs from *src* to *spr* (using *dst* as the temporary spare tower), then move one disc from *src* to *dst*, and finally move the $n - 1$ discs moved previously from *spr* to *dst* (using *src* as the temporary spare tower). Let's look at the first call only along with the function definition:

```
def hanoi(n, src, dst, spr):
    ...
    hanoi(n - 1, src, spr, dst)
```

In this call, the actual parameter, $n - 1$, is mapped to the formal parameter, *n*. So whatever *n* is in the function, when it is called again, it is with $n - 1$. Similarly, *src* is mapped to *src* (since moving *n* discs from some source results in moving $n - 1$ discs from that source first). The $n - 1$ discs, however, are moved to the spare tower (to get them out of the way). So the call maps *spr* as the actual parameter to *dst* as the formal parameter. Lastly, *dst* is mapped to *spr*. That is, the destination tower is temporarily used as the spare tower when moving the $n - 1$ discs out of the way.

Perhaps this is best viewed dynamically, as each call is made. Let's illustrate this with a simple puzzle with two discs:

```
hanoi(2, "A", "C", "B")
```

This call results in the following variable values:

n	*src*	*dst*	*spr*
2	A	C	B

The recursive step is then applied, which results in the following three recursive calls:

```
            1    A    B    C
hanoi(n - 1, src, spr, dst)
            A    C    B
hanoi(1, src, dst, spr)
            1    B    C    A
hanoi(n - 1, spr, dst, src)
```

Since these are all one-disc puzzles, the base case is applied each time (which simply displays the move to the console). The first recursive call results in the the following variable values:

n	*src*	*dst*	*spr*
1	A	B	C

It therefore displays the following move: A → B. The second recursive call results in the the following variable values:

n	*src*	*dst*	*spr*
1	A	C	B

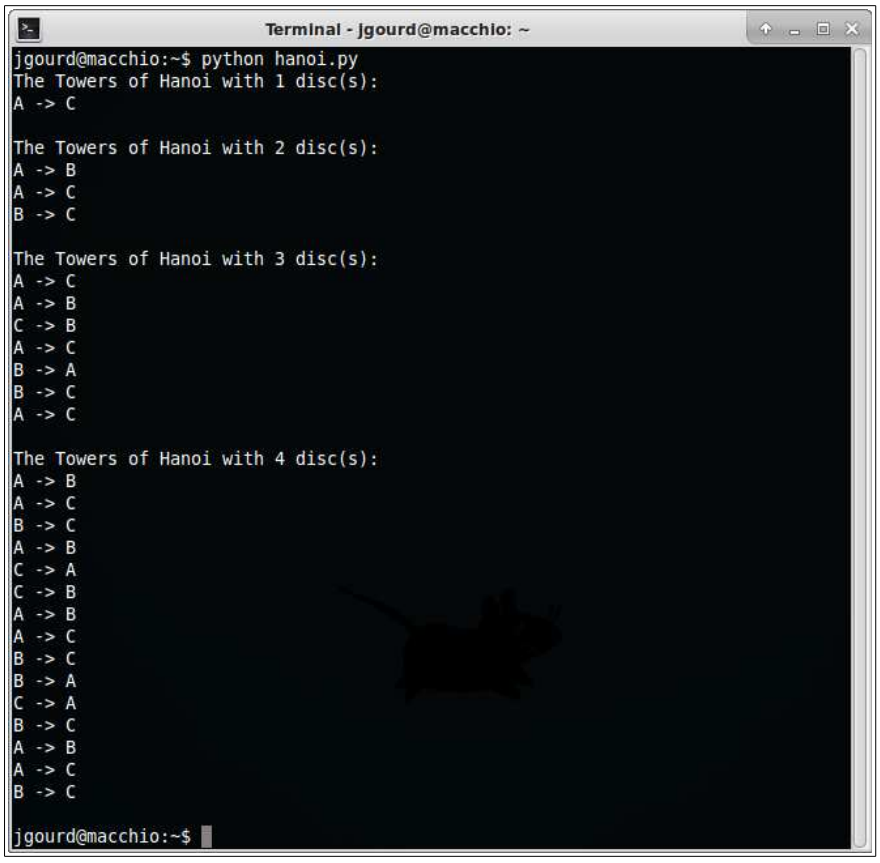
It therefore displays the following move: A → C. The third recursive call results in the the following variable values:

n	*src*	*dst*	*spr*
1	B	C	A

It therefore displays the following move: B → C. The three moves do, in fact, solve the two-disc puzzle. Note that the values for the three towers can be anything. The strings "A", "B", and "C" were used above; however, the integers 1, 2, and 3 would work just as well. If we wish, we can display the moves required for puzzles with one through four discs as follows:

```
for i in range(1, 5):
    print "The Towers of Hanoi with {} disc(s):".format(i)
    hanoi(i, "A", "C", "B")
    print
```

The output of this modified algorithm is:



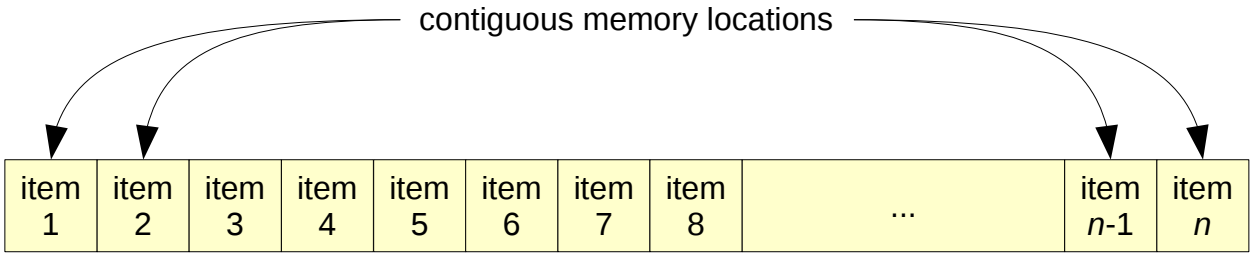
This lesson looks at the topic of data structures, which is concerned with the various ways that data can be organized within a computer program. Specifically, four common "high level" data structures are introduced: lists, stacks, queues, and trees. Each of these structures represents a way to organize data so that it may be applied to solve certain problems in an efficient manner. Three of these structures (lists, stacks, and queues) are linear in nature. That is, their items logically exist one after another in sequential order. The tree structure is non-sequential, in that its contents can not be meaningfully represented by a simple sequential listing.

Lists

When designing algorithms to solve computing problems, it is rare that we do not, to some degree, store and manipulate data. Often, we need to store data in a list form. This has been observed in previous lessons (e.g., searching a list for some value, sorting a list using the insertion sort, etc). In fact, we have seen (and used) lists in Scratch and in Python to solve problems.

In general, a list groups values together in such a way that there is a first item in the list, a last item in the list, and some number of items in the middle of the list. Accessing any individual item in the list is permitted (using its position or index). There are generally two implementations of lists: array-based lists and linked lists.

We have already seen arrays in a previous lesson; however, let's briefly review. **Arrays** are comparable to a numbered list such as a grocery list, a class roster, or a set of numbered drawers. They are used to store multiple instances of anything, as long as they are all of the same kind (i.e., all numbers, all letters, all images, all books, etc). Imagine these things being in some sort of order (i.e., we have a first thing, a last thing, and some number of things in between). The members of (or entries in) the array are called elements.



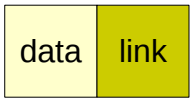
The order in which elements are stored in an array is important. This is because very often a programmer needs to access a specific element of an array, and in order to do that, its position relative to the first element of the array must be known. The position of an element is also referred to as its *address*, and the relative address (how far away from the first element it is) is called its *index*.

Again, the distinction between a value and its index is one that must be emphasized. A value refers to a piece of data stored in the array, and its index is the position in the array where that value is stored. The index represents *where* an element is, and the value represents *what* the element is. While the two are related, each of them will be of different importance to us depending on the scenario we are trying to solve.

In most programming languages, arrays must be declared along with their capacity (i.e., the maximum number of values that the array can contain). This is important because array elements are located in contiguous memory locations (i.e., next to each other in memory). Therefore, an array's capacity is required in order to properly allocate all of the contiguous memory locations needed. This does, however, represent the array's weakness. If we know how many items we will store in the array, then this is simple. But what if we don't? We may purposely overestimate, but this wastes memory. What if we underestimate and fill the array? The cost of creating a new, larger array, and then duplicating the existing array to this new array can be quite large. The need for specifying an array's capacity at declaration time is what motivated a tweak on implementing lists in programming languages. The result is the linked list.

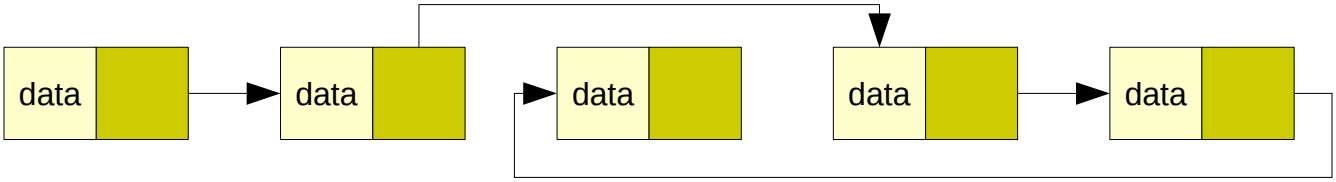
Unlike arrays, **linked lists** can grow or shrink as needed. Individual elements in a linked list are not necessarily stored in contiguous memory locations. The cost of this benefit is that some mechanism for *linking* each element in the list must exist. Ultimately, this mechanism requires additional memory; thus, linked lists require more space in memory than arrays do when storing the same data. However, this is offset by the convenience of only storing what is needed in the list at any one time.

Since the elements in a linked list are not necessarily stored in contiguous memory locations, we must somehow link the elements in the list to one another. This requires extending the concept of *element* to include two components: actual data (i.e., some value) and a link to the next element in the list. This extended definition makes up what is called a node. Linked lists are made up of **nodes**, each of which stores **data** and a **link** to the next node in the linked list:



If each node in the list contains some data and a link to the next node, then we really only need to know the location of the first node in order to process each element in the list. From the first node, we can repeatedly following links to the next node, until the end of the list is reached:

Of course, it does not necessarily have to look so pretty. The linked list above is drawn so that each element seems to be located in contiguous memory locations; however, we could just as easily have drawn the linked list like this:

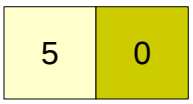


In this case, the first node is still the one all the way to the left. The last node, however, is now the third from the left.

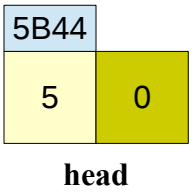
The first node in a linked list is known as its **head**. Knowing where the head is in a linked list is crucial.

So how do we know when to stop when, for example, we process each element in the list? That is, how do we know when we are at the last element in the list? The answer lies in the link component of the node. For clarity, you should know that the last node in a linked list is known as the **tail**. The link component of the tail of a linked list will always be *nothing*. In Python, we would say that its value would be equal to `None`. In other programming languages, we sometimes refer to this as *null*. For simplicity, the link component of the tail node is equivalent to 0.

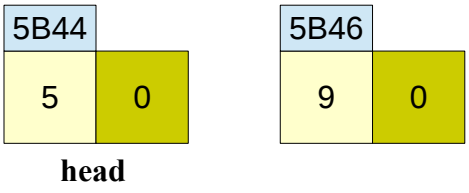
So what does the link component of a node other than the tail actually store? It stores the memory address of the next node in the list! Perhaps this is best explained by creating a linked list, step-by-step, and showing what happens at each step. Let's insert the values 5, 9, 2, 6, and 1 into a linked list. Inserting the value 5 first requires creating a new node with 5 as the data component and 0 as the link component (by default):



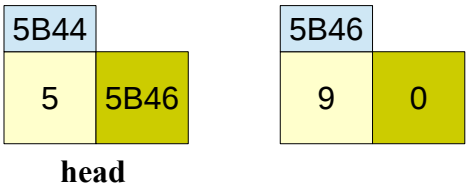
Of course, this node must be stored somewhere in memory. Let's randomly pick the memory address 5B44 (in hexadecimal). In addition, since this is the only node in the linked list so far, then it is at its head:



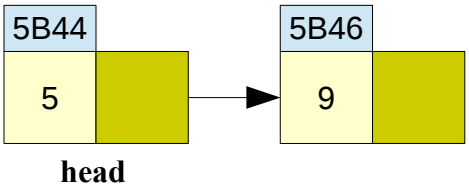
Suppose that inserting the value 9 creates a new node that is stored at memory address 5B46:



The nodes must now be linked. Since this new node belongs after the head of the list (i.e., it is the second node inserted into the list), then we simply need to link the head to this new node as follows:

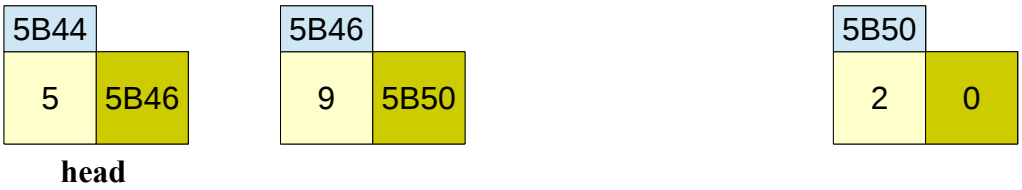


Notice how the link component of the head contains the memory address of the next node in the list. This has the same effect as the following pictorial example:



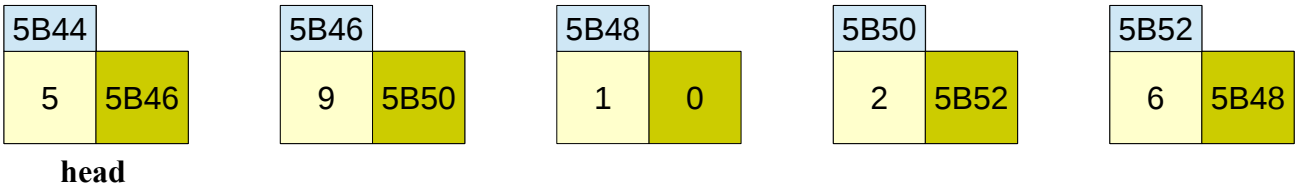
Since we know where the head of the linked list is located (by definition), then we can reach the second node by following the link (i.e., by moving the to memory address specified in the link component of the node).

Inserting the value 2 is similar. Suppose that the new node containing this value is stored at memory address 5B50:

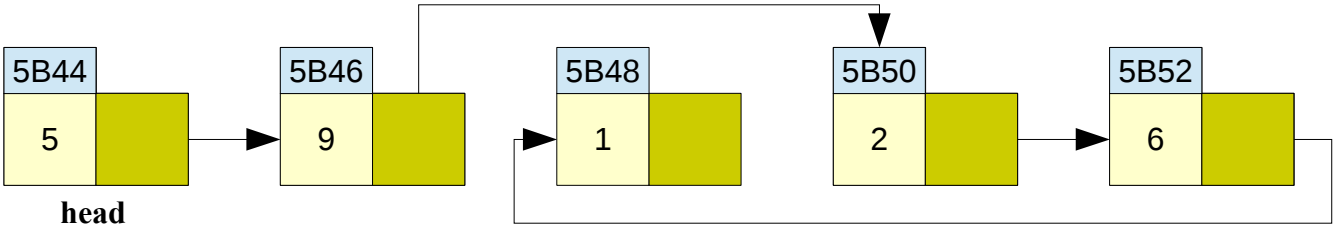


Notice how the link component of the node containing the value 9 correctly links it to the newly inserted node. Now let's insert the value 6. Suppose that the new node is stored at memory address 5B52:

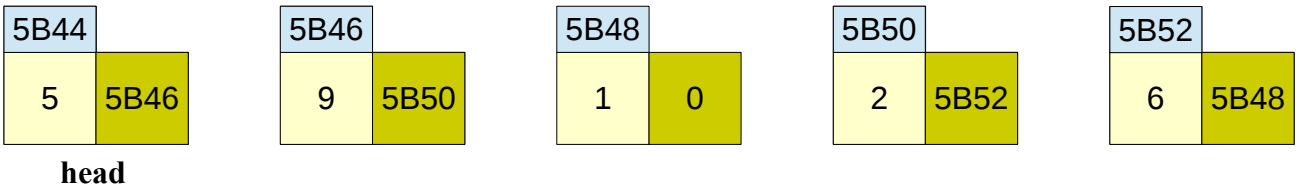
Finally, let's insert the value 1. Suppose that the new node is stored at memory address 5B48:



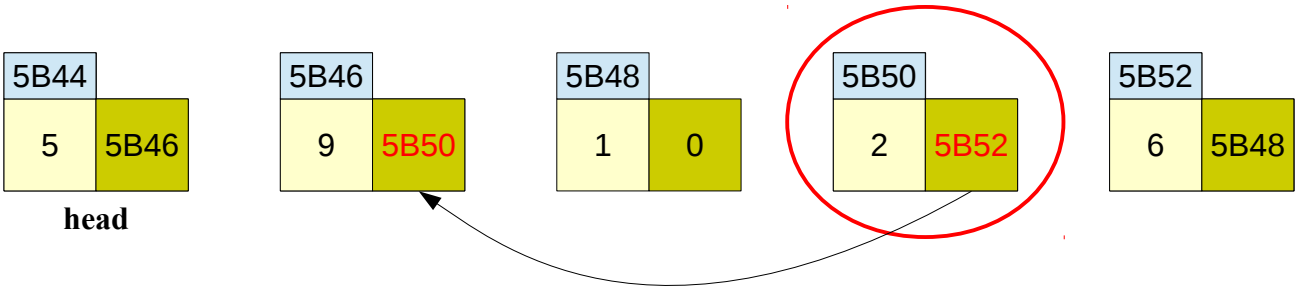
Notice how we can start at the head of the linked list and follow the link components of each node, all the way through the tail. We know to stop at the tail because its link component is 0. Pictorially, this can be represented as follows:



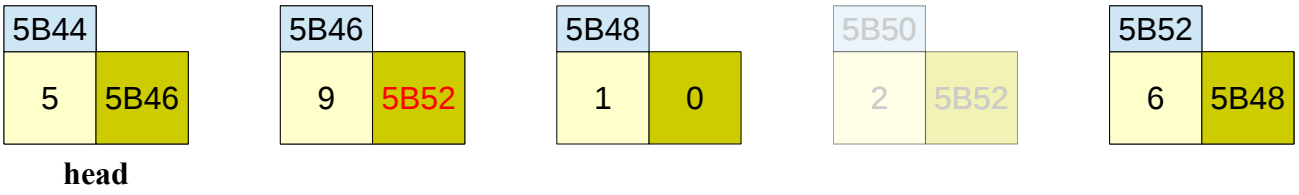
Just like arrays, linked lists can be used to implement searching and sorting algorithms. Processing each element is just as simple. Moving elements around, however, is a bit more complicated since it involves rearranging the link components of nodes to reflect a potentially new ordering of the elements in the list. For example, consider the problem of deleting the node containing the value 2:



Starting at the head, we see that the node containing the value 2 is the third node in the list. To remove this node, we will need to change the node containing the value 9 (since it is linked to the node that we wish to delete). The solution is to change the link component of this node so that it is equivalent to the link component of the node that we wish to delete:

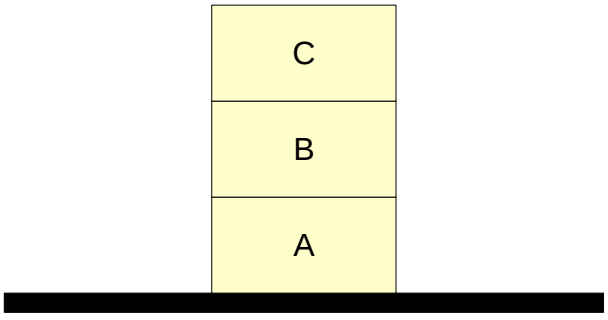


The node to be deleted is circled in red above. To remove it from the linked list, we simply need to copy its link component to the node that precedes it, thereby *rerouting* around it. After this action, the result is the following linked list (the grayed out node is no longer a part of the linked list):



Stacks

Stack data structures are used, among other things, to model the behavior of stacks of real-world objects. In order to understand this structure let's begin by thinking about a simple stack of blocks:



In this stack, **C** is the top block, **B** is beneath **C**, and **A** is the bottom block. Assuming that the blocks were added one at a time, how must this stack have been built? First, the bottom block, **A**, must have been set, then **B** would have been placed on top of **A**, and finally **C** would have been placed on top of **B**. Note that the blocks must be added to the stack from bottom to top: **A**, then **B**, then **C**. The last block placed on the stack will be the top block.

Now, let's think about removing a block from the stack. As any child could demonstrate, the block that is most easily accessible is the top block (**C** in this case). Removing **C** from the stack leaves us with **B** sitting on top of **A**. It is interesting to note that the first item to be removed from the stack, **C**, was the last item added to the stack (remember the order in which the blocks were added). In fact, assuming that you don't cheat and grab an item from the middle of the stack, the last item added on the stack would always be the first taken off. For this reason, stacks are known as Last-In, First-Out (LIFO) data structures.

As mentioned earlier, the stack data structure models the behavior of real-world stacks of objects. The two primary operations that can be applied to stack data structures are **push** and **pop**. The **push** operator is used to add a new item onto the top of the stack. The **pop** operator is used to remove the top item from the stack. Stack data structures do not support the removal of items from the middle of the stack. The stack of real-world blocks shown above could be modeled by applying the following operations to an initially empty stack data structure:

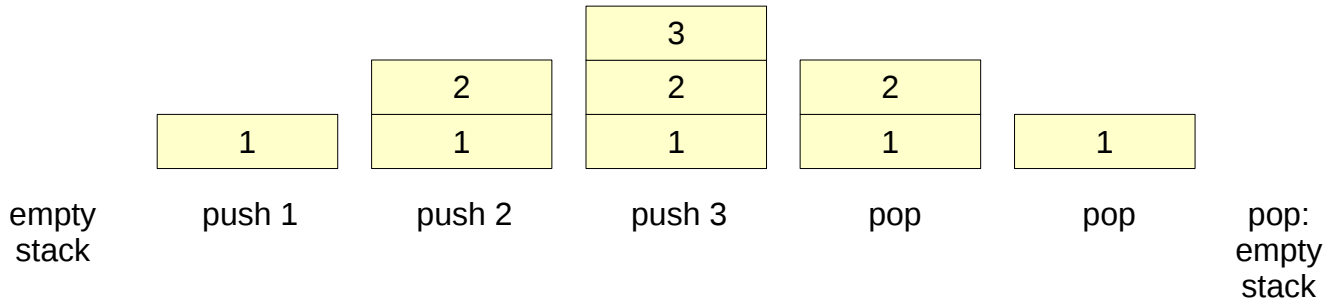
```
push A
push B
push C
```

The removal of **C** could subsequently be accomplished by issuing the pop command:

```
pop
```

To be more formal, the **stack** data structure can be defined as a specialized type of list (an ordered sequence of items) in which all insertions to and deletions from the list take place at one end. The end of the list where the insertions and deletions are performed is known as the **top** of the stack. Stacks are usually drawn vertically, so that the item at top of the stack appears literally as the topmost item in the structure; however, they could just as easily be drawn sideways or upside down.

In order to ensure that you have a clear understanding of the behavior of the stack data structure, consider the following sequence of stack operations and pictorial representations of the stack that would result after each operation is applied. Note again that the pop operation always removes the last item placed on the top of the stack:



Now that you have some understanding of how stacks behave, it is natural to ask, "So what?" Why are

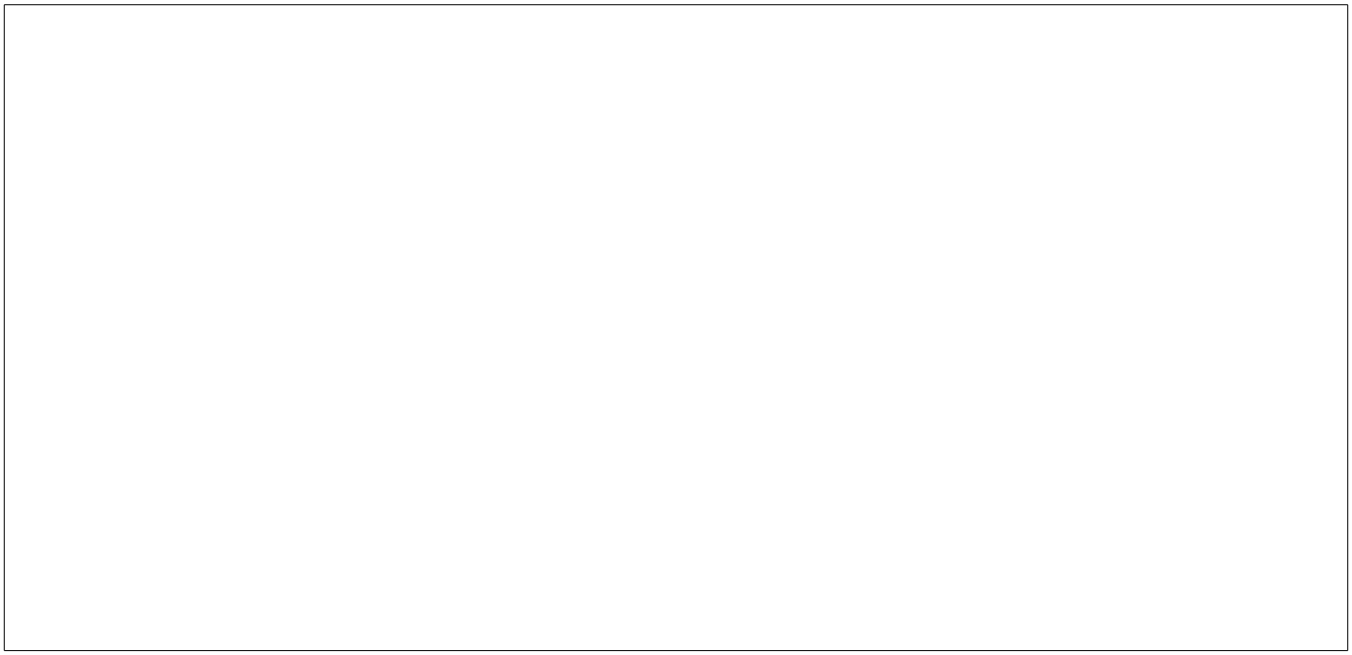
stacks of interest to computer scientists? In the real world, we routinely encounter stacks of objects (e.g., CDs, dishes, bills). We use these stacks to temporarily hold objects until we are ready to use or *process* them in some way. One important characteristic of all stacks (both the real-world type and their software counterpart) is that, due to their LIFO nature, they reverse the order of the objects they hold. For example, if you place three CDs in a stack, AC/DC, then Iron Maiden, then Justin Bieber, and then play the top one, you will unfortunately be listening to Justin Bieber and not AC/DC. In everyday life, we tend to use stacks in situations where order is unimportant (e.g., for holding identical, non-perishable items like dinner plates).

In a similar manner, stack data structures are used by computer software to temporarily hold data *objects* until they can be processed by the computer. However, instead of deemphasizing the LIFO nature of stacks, in computing stacks tend to be used almost exclusively in situations where we specifically want to process items in the opposite order than they were added to the structure.

One common use of stacks in computing is to manage the execution of interruptible tasks. The utility of stacks for this purpose can easily be seen by a real-world analogy. Say that you are typing an English paper (task one) and the phone rings. You pick up the phone (i.e., place task one on the stack of *on hold* processes) and begin a conversation with a friend (task two). During this conversation you get a second phone call, so you put your friend on hold (i.e., place task two on the stack of *on hold* processes) and take a call from your mom (task three). After a brief talk with mom, you switch back to your friend (i.e., after task three completes, you **pop** task two off of the stack of *on hold* processes and restart it from the point you left off). Finally, after a not-so-brief conversation with your friend, you return to your English paper (i.e., after task two completes, you **pop** task one off of the stack of *on hold* processes and restart it where you left off).

Note that in order to handle these interruptible tasks properly, a data structure such as a stack that incorporates LIFO behavior must be employed.

Try your hand at **push**ing the letters of the word PUPILS, one letter at a time, to a stack in the space below:



Now **pop** each letter off of the stack, one at a time, in the space below. While doing so, record each **pop**ped letter to see the word formed after the **pop** operations are complete and the stack is empty again:

As you can see, a stack can easily reverse a word. We can also use a stack to match parentheses in, for example, mathematical expressions. The basic idea is to scan through an expression, one character at a time, from left-to-right. Left (or open) parentheses are pushed on the stack. Right (or close) parentheses result in a pop (and a match of the left parenthesis that was just popped). Operators and operands are ignored.

Take a look at the following expression and its resulting stack operations. To make it easier to follow, we'll change the orientation of the stack so that the top is to the right:

```
a + b * (c + (d - e) / (f / g))
```

Input	Operation	Stack (top →)
a	ignore	
+	ignore	
b	ignore	
*	ignore	
(	push	(
c	ignore	(
+	ignore	(
(	push	((
d	ignore	((

Input	Operation	Stack (top →)
`-`	ignore	`( (`
`e`	ignore	`( (`
`)`	pop (and match)	`(`
`/`	ignore	`(`
`(`	push	`( (`
`f`	ignore	`( (`
`/`	ignore	`( (`
`g`	ignore	`( (`
`)`	pop (and match)	`(`
`)`	pop (and match)	

So long as the stack is empty at the end, all parentheses have been matched. Create the table for the following expression:

```
(a + (b - c)
```

Input	Operation	Stack (top →)

The error arises because the expression has been processed, but there is still an open parenthesis on the stack.

Now create the table for the following expression:

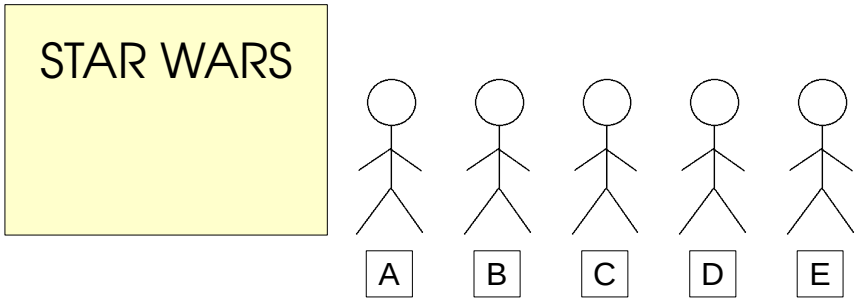
```
(a + b))
```

Input	Operation	Stack (top →)

The error arises because there is a close parenthesis left to process, but there is no matching open parenthesis on the stack (it is empty).

Queues

In America, a waiting line (such as the kind you encounter at a bank or supermarket), is simply referred to as a *line*. In England and many other countries, a waiting line is called a *queue* (pronounced like the letter Q). The following illustrates a real-world queue that you might encounter at a local movie theater:



Person A is buying a ticket to see the next Star Wars movie. Persons B, C, D, and E are waiting in line to buy their tickets. The four of them are considered to be waiting in the queue. Since person A is in the process of buying his ticket, he is not considered to be part of the queue (i.e., he is not waiting). Person B is at the front, or head, of the queue, and will be the next person to be served. Person E is at the back, or end, of the queue, and must wait for everyone ahead to be served before being able to buy a ticket.

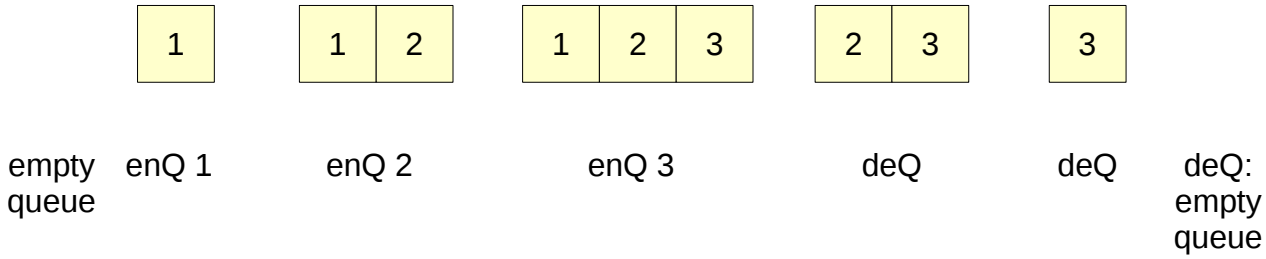
Queues are known as First-In, First-Out (FIFO) data structures. Assuming that no one breaks or cuts in line, the first person to enter the queue will be the first person to leave the queue (and thus be the first person served). Queues are extremely useful, both in the real-world and in computing, because they enable us to control access to scarce resources (such as Star Wars tickets in the example above).

The queue data structure models the behavior of real-world queues. It supports two primary operations: **enqueue** and **dequeue**. The **enqueue** operation is used to add a new item to the back of the queue. This operator is analogous to a person getting in line. The **dequeue** operation is used to remove an item from the front of the queue. This operation is analogous to having the person at the head of the line step forward to be served (and then of course to purchase grossly overpriced and over-buttered popcorn and a

soda).

More formally, a **queue** is a list in which all insertions take place at one end, the **back** of the queue, and all deletions take place at the opposite end, the **front** (or head) of the queue. The basic queue data structure does not allow the insertion or deletion of items from the middle. Thus, it does not support concepts like people breaking or cutting in line, or giving up and leaving because the line to too long.

Consider the following sequence of operations applied to an initially empty queue, and the resulting queue configurations. In order to help clearly distinguish between the behavior of stacks verses queues, the same data presented in the (numeric) stack example above is used (along with the same pattern of operations applied to the structure). Notice that, even though the order of operations is the same in both of these examples, the contents of the two structures is quite different:

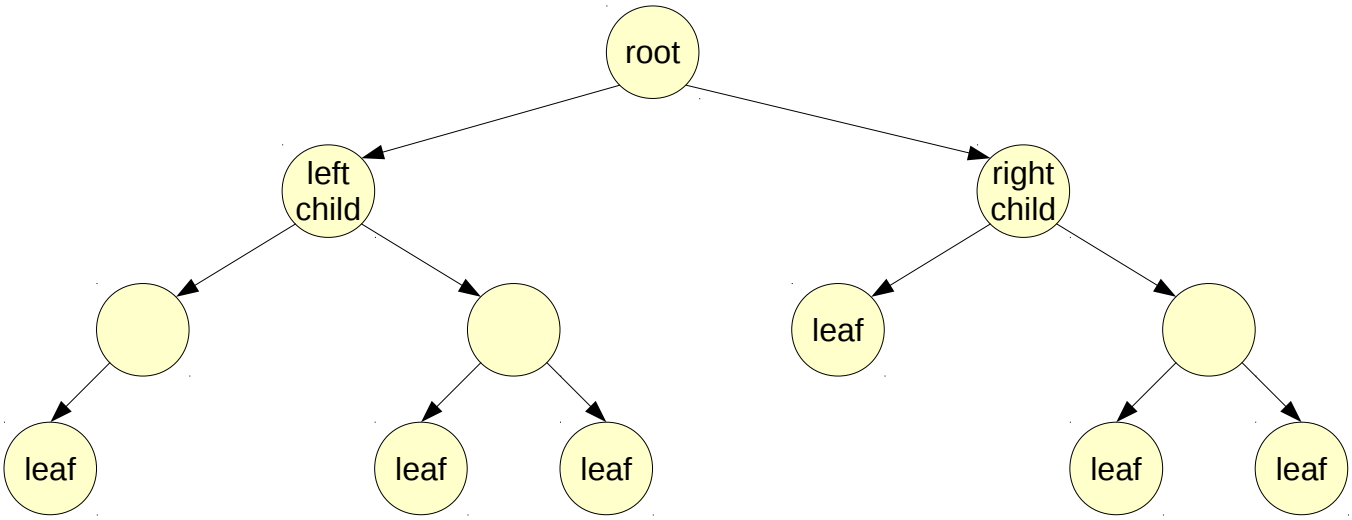


Trees

Although there are many different types of tree-like data structures, we will focus on the **binary tree** in this lesson. First, let's recall the various searching algorithms that were discussed in previous lessons. One in particular was especially efficient on sorted data: the binary search. The binary search works well because it effectively halves the search space with each comparison. The strategy is to continuously pick the middle value in a portion of the list making up half of what is left to search through. Initially, the middle of the list is picked. If the specified value is not found, then the half of the list that it cannot be contained within is discarded, and the process repeats with the other half of the list. This idea of halving the search space with a single comparison is the foundation for the binary tree.

Earlier, you learned that linked lists were made up of nodes that point to other nodes in a linear or sequential fashion. These other nodes could be considered neighbors. Reaching the nodes in a linked list is performed by following the link component of each node until the tail of the list is reached. One could maintain a sorted linked list and implement the binary search, for example. Binary trees are also made up of nodes; however, instead of a node containing a link to some next *neighbor* node, each node in a binary tree contains two link components. These link components are known as the node's children. This represents a different relationship between nodes.

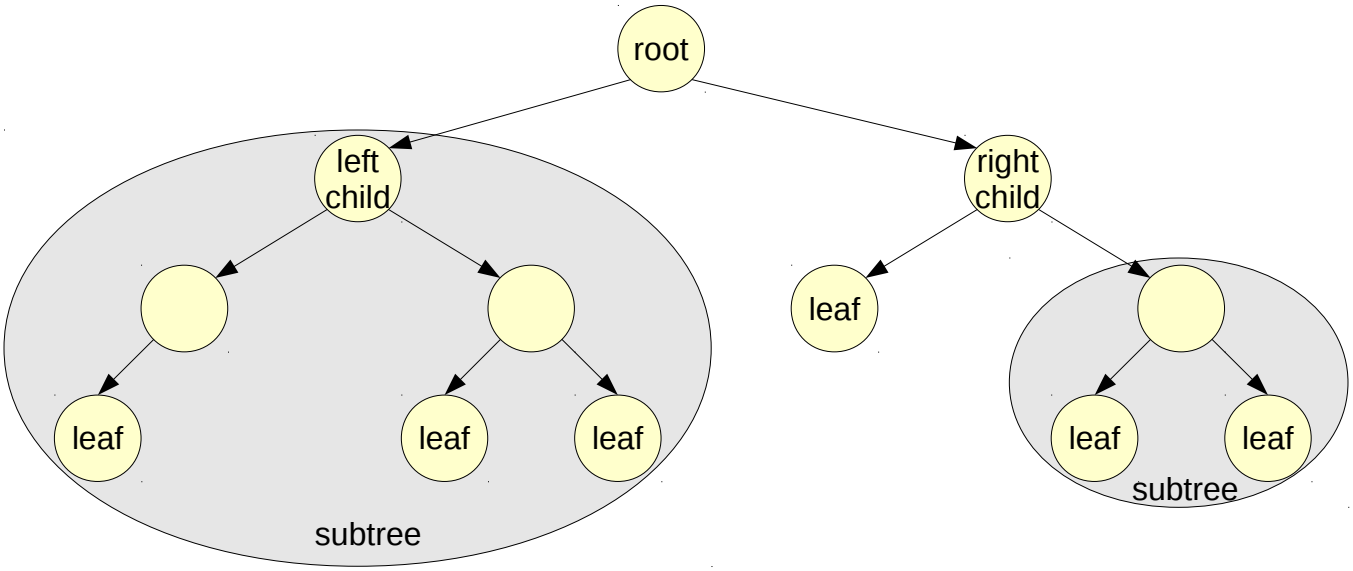
Perhaps it is best to see what a tree actually looks like for reference:



The top node in the tree is known as the **root** of the tree. In a binary tree, each node has up to two **children**. In the tree above, the root node has two children (one to the left, and one to the right – which is usually where we place them). In turn, these two children have children of their own. This can continue many times; however, at some point there will be nodes at the bottom of the tree that do not have children. These nodes are known as **leaf** nodes. Collectively, they are referred to as the **leaves** of the tree.

Why is this data structure called a binary tree? Well, if we were to turn the image above upside-down, it would look kind of like a real tree with a root at the bottom that splits into branches. Each branch splits into more, and so on, until the leaves are reached. It is called a **binary tree** because each node in the tree has, at most, two children. As branches are followed down the tree, the tree repeatedly splits into two halves, much like the process behind the binary search.

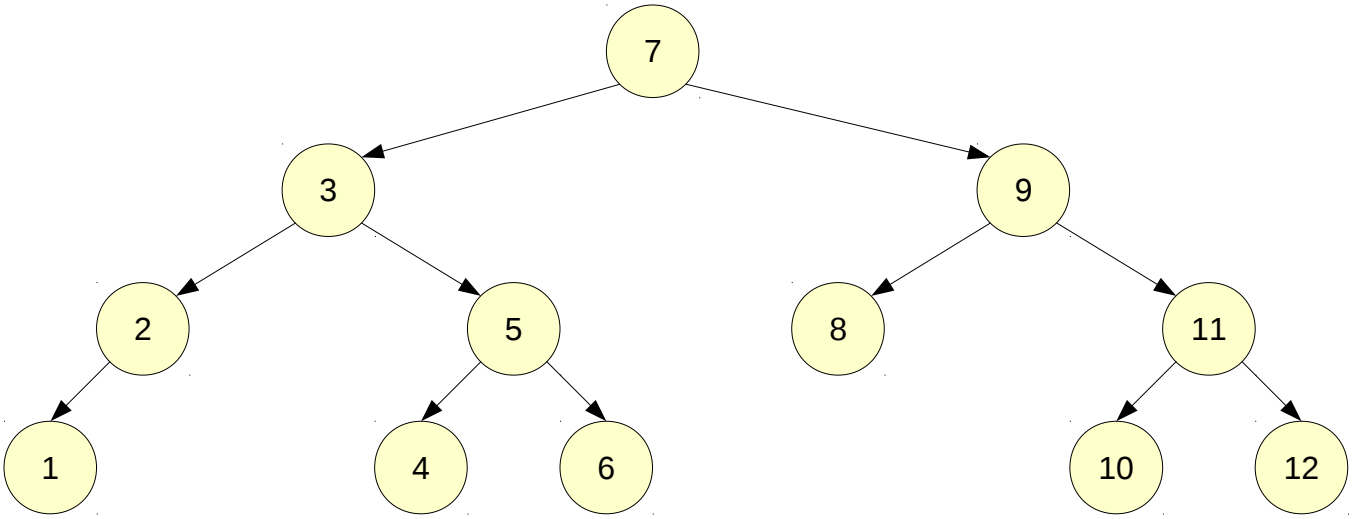
Sometimes it is useful to only consider parts of the tree. These parts are known as **subtrees**:



Although there is only a single root in a binary tree (always the node at the top of the tree), each subtree can be said to have a node that serves as its root. For example, the node labeled "left child" in the tree above can be said to be the root of the shaded subtree on the left.

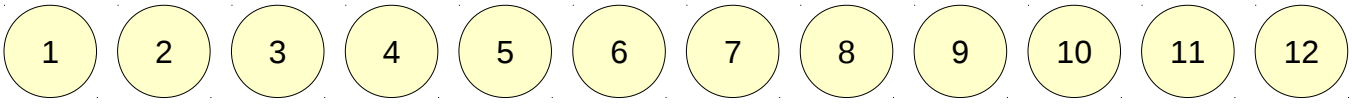
The nodes in a binary tree are arranged in levels. A **level** can be said to be a horizontal *slice* of the binary tree. In the tree above, for example, the nodes labeled "left child" and "right child" are at the same level. In fact, they are at level 1. The root is at level 0. For a binary tree that is **balanced** (i.e., one that isn't lopsided with too many extra nodes on any side), the number of levels is a function of the number of nodes. In a binary tree containing n nodes, there can be, at most, $\lceil \log_2 n \rceil$ levels. Intuitively, this makes sense since each level down the tree splits that part of the tree in half. In the tree above, there are 12 nodes; thus, there can be at most $\lceil \log_2 12 \rceil = \lceil 3.58 \rceil = 4$. In fact, there are four levels in the tree!

The most useful binary trees implement an internal ordering of the nodes. **Ordered binary trees** are binary trees that abide by the rule that, given any node, the values of all children in the left subtree are less than the value of the node. Similarly, the values of all children in the right subtree are greater than (or sometimes greater than or equal to) the value of the node. Here is an example of an ordered binary tree:



Notice that, for any node in the tree, the values of all children in its left subtree are less than the value of the node. Similarly, the values of all children in its right subtree are greater than the value of the node. For example, the values of all nodes to the left of the root are all less than seven, and the values of all nodes to the right of the root are all greater than 7.

If we were to place all nodes at the same level (i.e., all next to each other) and remove all links to children, this binary tree would look very much like a sorted array:



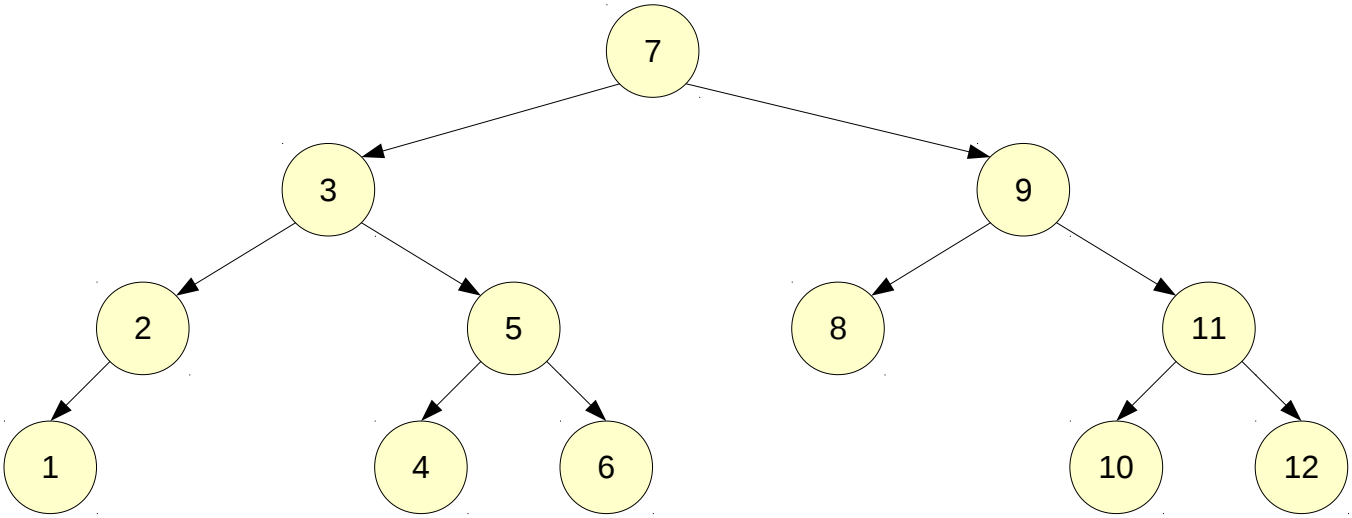
Notice that we can easily implement a binary search! An ordered binary tree actually works very much the same way. To search for a value, we begin at the root and compare its value to the desired value. If the desired value is less than the value of the root, we follow the link to the left and continue. If the desired value is greater than the value of the root, we follow the link to the right and continue. This

exactly replicates the behavior of the binary search. If we find a node that is equal to the desired value, then the search is successful. If we reach a leaf node that is not equal to the desired value, then the search is unsuccessful.

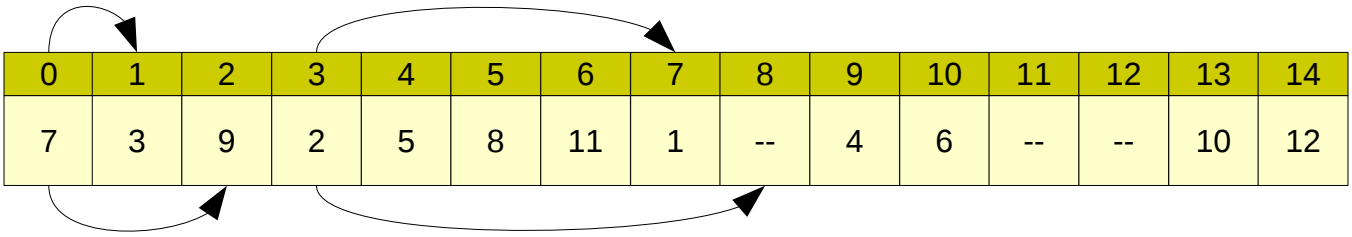
An interesting aspect of the binary tree is that we can use an array to represent one in memory. The root of the tree is placed at the first position in the array (i.e., at index 0). The children of any node are easily found as follows:

- The left child of a node at index i is at index $2i+1$;
- The right child of a node at index i is at index $2i+2$; and
- The parent of a node at index i is at index $\lfloor (i-1)/2 \rfloor$.

Take a look at the previous tree again:



This tree can be represented by an array as follows:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	3	9	2	5	8	11	1	--	4	6	--	--	10	12

The arrows are there to illustrate a few of the nodes and their children. For example, the root node is at index 0. Its children are at indexes $2(0)+1=1$ and $2(0)+2=2$. This is illustrated by the arrows pointing from the root node in the array. As another example, the children of the node containing the value 2 (which is at index 3) are at indexes $2(3)+1=7$ and $2(3)+2=8$. Nodes with missing children are still included in the array (i.e., space is left for the missing children). As a final example, the parent of the node containing the value 6 (which is at index 10) is at index $\lfloor (10-1)/2 \rfloor = \lfloor 9/2 \rfloor = \lfloor 4.5 \rfloor = 4$. These are all confirmed in the tree above.

RPi Activities

Raspberry Pi Activity: Room Adventure

In this activity, you will implement a simple text-based game utilizing the object-oriented paradigm. You will need the following items:

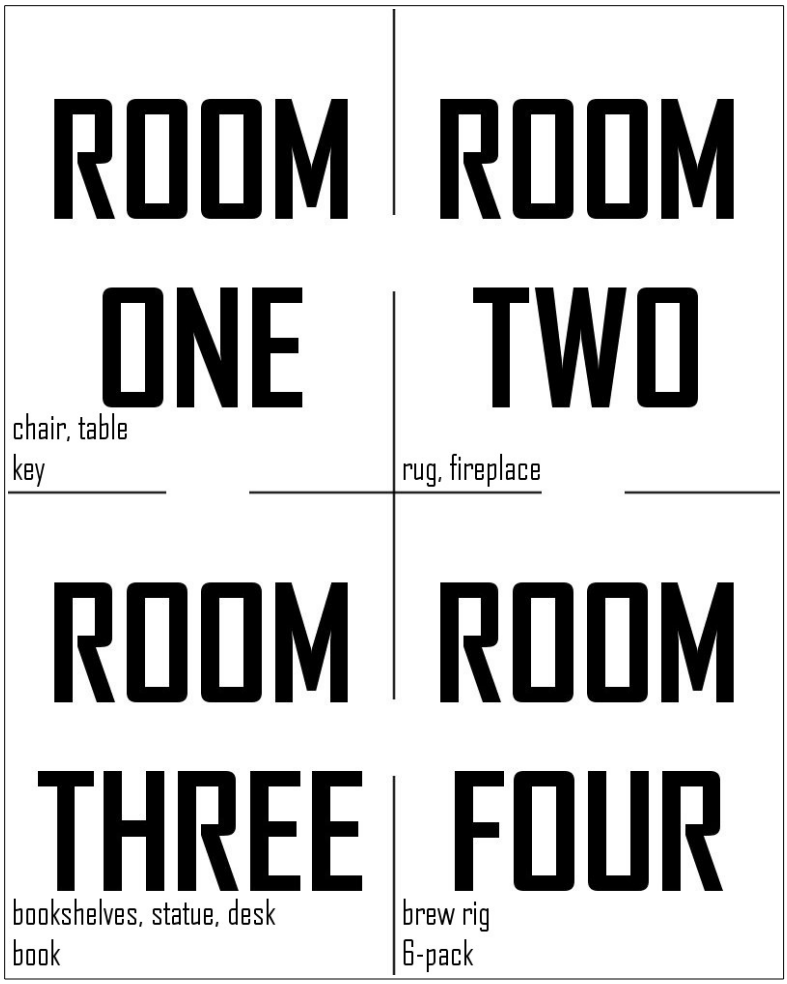
- Raspberry Pi B v3 with power adapter;
- LCD touchscreen; and
- Keyboard and mouse.

If you wish, you can simply bring your laptop with the Python interpreter (and also perhaps IDLE) installed since you will not be using the GPIO pins on the RPi.

The game

The first step to designing any kind of computer program is to establish the goal or purpose and to identify all of the necessary details. This is very similar to first understanding a problem before setting about to solve it. Although we will build a very simple game, it can still have (very simple) goals.

The setting of the game is a small "mansion" consisting of four rooms. Here's the layout:



Each room has various exits that lead to other rooms in the mansion. In addition, each room has items, some of which can simply be observed, and others that can be picked up and added to the player's inventory. For this activity, there is no actual *goal* for the player other than to move about the mansion, observe various items throughout the rooms in the mansion, and add various items found in the rooms to inventory. There is an end state that results in death, however! Of course, this doesn't prevent extending the game with a better story and more variety (some ideas will be discussed later).

The four rooms are laid out in a simple square pattern. Room 1 is at the top-left of the mansion, room 2 is at the top-right, room 3 is at the bottom-left, and room four is at the bottom-right. Each room has items that can be observed:

- Room 1: A chair and a table;
- Room 2: A rug and a fireplace;
- Room 3: Some bookshelves, a statue, and a desk; and
- Room 4: A brew rig (you know, to brew some delicious libations).

Observable items can provide useful information (once observed) and may reveal new items (some of which can be placed in the player's inventory). In addition, each room may have some items that can be *grabbed* by the player and placed in inventory:

- Room 1: A key;
- Room 3: A book; and
- Room 4: A 6-pack of a recently brewed beverage.

The rooms have various exits that lead to other rooms in the mansion:

- Room 1: An exit to the east that leads to to room 2, and an exit to the south that leads to room 3;
- Room 2: An exit to the south that leads to room 4, and an exit to the west that leads to room 1;
- Room 3: An exit to the north that leads to room 1, and an exit to the east that leads to room 4; and
- Room 4: An exit to the north that leads to room 2, an exit to the west that leads to room 3, and an (unlabeled) exit to the south that leads to...death! Think of it as jumping out of a window.

The gameplay

The game is text-based (egads, there are no graphics!). Situational awareness is provided by means of meaningful text that describes the current situation in the mansion. Information such as which room the player is located in, what objects are in the current room, and so on, is continually provided throughout the game. The player is prompted for an action (i.e., what to do) after which the current situation is updated.

The game supports a simple vocabulary for the player's actions that is composed of a verb followed by a noun. For example, the action “go south” instructs the player to take the south exit in the current room (if that is a valid exit). If the specified exit is invalid (or, for example, if the player misspells an action), an appropriate response is provided, instructing the player of the accepted vocabulary. Supported verbs are: *go*, *look*, and *take*. Supported nouns depend on the verb; for example, for the verb *go*, the nouns *north*, *east*, *south*, and *west* are supported. This will allow the player to structure the following *go* commands:

- go north
- go east

- go south
- go west (young man!)

The verbs *look* and *take* support a variety of nouns that depend on the actual items located in the rooms of the mansion. The player cannot, for example, "look table" in a room that doesn't have a table! Some examples of *look* and *take* actions are:

- look table
- take key

The gameplay depends on the user's input. Rooms change based on valid *go* actions, meaningful information is provided based on valid *look* actions, and inventory is accumulated based on valid *take* actions. For this game, gameplay can continue forever or until the player decides to "go south" in room 4 and effectively jump out of the window to his/her death:

At any time, the player may issue the following actions to leave the game:

- quit
- exit
- bye

The design

By now, you may have already noticed that the four rooms are similar. They each have exits, items that may be observed, and items that may be added to the player's inventory. A good design choice is to utilize the object-oriented paradigm and implement a class that serves as the blueprint for all of the rooms in the game. In fact, let's begin there. Launch IDLE or a text editor and edit a new Python program. Call the document **RoomAdventure.py**. Let's begin with an informative header and the beginning of the room class:

```
#######################################################################
# Name:
# Date:
# Description:
#######################################################################

#######################################################################
# the blueprint for a room
class Room(object):
     # the constructor
```

```
def __init__(self, name):
     # rooms have a name, exits (e.g., south), exit locations
     # (e.g., to the south is room n), items (e.g., table), item
     # descriptions (for each item), and grabbables (things that
     # can be taken into inventory)
     self.name = name
     self.exits = []
     self.exitLocations = []
     self.items = []
     self.itemDescriptions = []
     self.grabbables = []
```

Note that it is wise to periodically save your program in case something happens. Also, it may be best to simply read the source code in this activity so that you understand what is going on instead of actually writing the code as it appears in the document. The entire code will be shown later so that you can write it in its entirety and see the full context.

So far, we have covered the constructor of the room class. It does what most constructors do: initializes various instance variables. In this case, a room object must be provided a name in order to be successfully instantiated (note the `name` parameter in the constructor). The room's name is assigned, and lists that contain its exits, observable items, and items that can be placed in inventory are initialized (all as empty lists). The variables are defined as follows:

- *name*: contains the room's name;
- *exits*: contains the room's exits (e.g., north, south);
- *exitLocations*: contains the rooms found at each exit (e.g., to the south of this room is room 2);
- *items*: contains the observable items in the room;
- *itemDescriptions*: contains the descriptions of the observable items in the room; and
- *grabbables*: contains the items in the room that can be placed in inventory.

Note that the lists *exits* and *exitLocations* are known as parallel lists. So are the lists *items* and *itemDescriptions*. **Parallel lists** are two or more lists that are associated with each other. That is, they have related data that must be combined to provide meaningful information. The lists are correlated by position (or index). That is, the items at the first index of *exits* and *exitLocations* form a meaningful pair. If this were the object reference for room 1, for example, the first item in the list *exits* could be the string "east". Consequently, the first item in the list *exitLocations* would then be the object reference for room 2, since moving east from room 1 should lead to room 2. Similarly, the first item in the list *items* for room 1 could be "table". Consequently, the first item in the list *itemDescriptions* would then be "It is made of oak. A golden key rests on it." (since this is the description for the table in room 1).

The room class should then provide appropriate getters and setters for each instance variable. Recall that this is accomplished by providing getter and setter methods that manipulate (either access or change) instance variables, each of which typically begins with an underscore. For example, the getter for the list *exits* would be a function named `exits` that would return the instance variable (a list) *_exits*. Here are the getters and setters for each instance variable. These are located in the room class, beneath the constructor (make sure to properly indent!):

```
# getters and setters for the instance variables
@property
```

```
def name(self):
     return self._name

@name.setter
def name(self, value):
     self._name = value

@property
def exits(self):
     return self._exits

@exits.setter
def exits(self, value):
     self._exits = value

@property
def exitLocations(self):
     return self._exitLocations

@exitLocations.setter
def exitLocations(self, value):
     self._exitLocations = value

@property
def items(self):
     return self._items

@items.setter
def items(self, value):
     self._items = value

@property
def itemDescriptions(self):
     return self._itemDescriptions

@itemDescriptions.setter
def itemDescriptions(self, value):
     self._itemDescriptions = value

@property
def grabbables(self):
     return self._grabbables

@grabbables.setter
def grabbables(self, value):
     self._grabbables = value
```

The approach in the game design will be to create an instance of the room class for each room in the mansion. With what has been implemented so far, the room blueprint exists, but there is currently no way to add exits, items, or grabbables. We must therefore provide methods to enable adding each of these things to a room. The first obvious addition is to provide support for adding exits (and their appropriate exit locations). Let's call this method `addExit`, and have it be provided with an exit and associated room as parameters:

```
# adds an exit to the room
# the exit is a string (e.g., north)
# the room is an instance of a room
def addExit(self, exit, room):
     # append the exit and room to the appropriate lists
     self._exits.append(exit)
     self._exitLocations.append(room)
```

The next obvious addition is to provide support for adding observable items and their associated descriptions. Let's do this similarly to the previous method (i.e., it will be provided with an item and its description) and call the method `addItem`:

```
# adds an item to the room
# the item is a string (e.g., table)
# the desc is a string that describes the item (e.g., it is made
# of wood)
def addItem(self, item, desc):
     # append the item and description to the appropriate lists
     self._items.append(item)
     self._itemDescriptions.append(desc)
```

Another obvious addition is to provide support for adding grabbables to the room in a method called `addGrabbable`. Since grabbables have no associated information, a single list is maintained. In addition, the method will be provided the grabbable item's name:

```
# adds a grabbable item to the room
# the item is a string (e.g., key)
def addGrabbable(self, item):
     # append the item to the list
     self._grabbables.append(item)
```

Since a grabbable item can be *grabbed* and placed in inventory, a method that removes the item from the room once it has been added to the player's inventory must be implemented. We will call this method `delGrabbable` and provided it with the item's name:

```
# removes a grabbable item from the room
# the item is a string (e.g., key)
def delGrabbable(self, item):
     # remove the item from the list
     self._grabbables.remove(item)
```

Lastly, it will be quite useful to provide a meaningful description of the room. This will make it simple to display all of the relevant information about the current room (e.g., exits, observable items, grabbables, etc). Recall that classes may specify a `__str__()` function that returns a string

representation of a class. Statements that *print* an instance of the class are then directed to this function for the appropriate string to display. Add the following function to the room class:

```
     # returns a string description of the room
     def __str__(self):
          # first, the room name
          s = "You are in {}.\n".format(self.name)

          # next, the items in the room
          s += "You see: "
          for item in self.items:
               s += item + " "
          s += "\n"

          # next, the exits from the room
          s += "Exits: "
          for exit in self.exits:
               s += exit + " "

          return s
```

Note that the escaped character, `\n`, adds a linefeed to the string (which means to go to the next line). Strings can span multiple lines! The function *builds* a string (the variable *s* in the function above). If the player is currently in room 1 at the start of the game, for example, the string would be formatted as follows:

```
     You are in Room 1.
     You see: chair table
     Exits: east south
     You are carrying: []
```

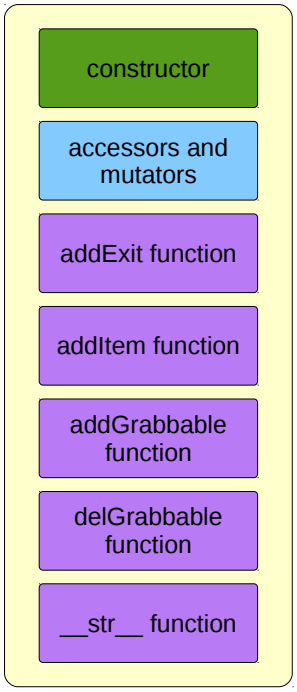
At this point, we are finished with the room class. For clarity, here it is in its entirety:

```
##############################################################################
# the blueprint for a room
class Room(object):
     # the constructor
     def __init__(self, name):
          # rooms have a name, exits (e.g., south), exit locations
          # (e.g., to the south is room n), items (e.g., table), item
          # descriptions (for each item), and grabbables (things that
          # can be taken into inventory)
          self.name = name
          self.exits = []
          self.exitLocations = []
          self.items = []
          self.itemDescriptions = []
          self.grabbables = []
```

```
    # getters and setters for the instance variables
    @property
    def name(self):
         return self._name

    @name.setter
    def name(self, value):
         self._name = value

    @property
    def exits(self):
         return self._exits

    @exits.setter
    def exits(self, value):
         self._exits = value

    @property
    def exitLocations(self):
         return self._exitLocations

    @exitLocations.setter
    def exitLocations(self, value):
         self._exitLocations = value

    @property
    def items(self):
         return self._items

    @items.setter
    def items(self, value):
         self._items = value

    @property
    def itemDescriptions(self):
         return self._itemDescriptions

    @itemDescriptions.setter
    def itemDescriptions(self, value):
         self._itemDescriptions = value

    @property
    def grabbables(self):
         return self._grabbables

    @grabbables.setter
    def grabbables(self, value):
         self._grabbables = value
```

```
# adds an exit to the room
# the exit is a string (e.g., north)
# the room is an instance of a room
def addExit(self, exit, room):
     # append the exit and room to the appropriate lists
     self._exits.append(exit)
     self._exitLocations.append(room)

# adds an item to the room
# the item is a string (e.g., table)
# the desc is a string that describes the item (e.g., it is made
# of wood)
def addItem(self, item, desc):
     # append the item and exit to the appropriate lists
     self._items.append(item)
     self._itemDescriptions.append(desc)

# adds a grabbable item to the room
# the item is a string (e.g., key)
def addGrabbable(self, item):
     # append the item to the list
     self._grabbables.append(item)

# removes a grabbable item from the room
# the item is a string (e.g., key)
def delGrabbable(self, item):
     # remove the item from the list
     self._grabbables.remove(item)

# returns a string description of the room
def __str__(self):
     # first, the room name
     s = "You are in {}.\n".format(self.name)

     # next, the items in the room
     s += "You see: "
     for item in self.items:
          s += item + " "
     s += "\n"

     # next, the exits from the room
     s += "Exits: "
     for exit in self.exits:
          s += exit + " "

     return s
```

For clarity, here's a layout of the Room class:



The main part of the game

It is now time to implement the main part of the game. Note that the room class discussed above only specifies what rooms are (i.e., their state and behavior). No rooms have been created thus far, no user input capability has been implemented, no decision-making based on user input has been implemented, and so on. This is our next task.

Let's begin by initializing an empty inventory list and creating the rooms of the mansion. The source code that follows belongs beneath the room class:

```
##############################################################################
# START THE GAME!!!
inventory = [] # nothing in inventory...yet
createRooms()  # create the rooms
```

Note the call to the function `createRooms()`. We often implement the main part of a program as a driver, in that it drives actions to be taken. Often, the actions are encapsulated in functions. Therefore, the main part of a program may call many functions. Overall, this helps to improve the readability of our programs and helps to simplify maintaining and updating them as well. The `createRooms()` function creates each instance of the four rooms. It sets their name, exits (and exit locations), observable items (and descriptions), and grabbable items. Lastly, it sets the player's current room at the beginning of the game (room 1). Let's work on the `createRooms()` function:

```
##############################################################################
# creates the rooms
def createRooms():
```

```
    # r1 through r4 are the four rooms in the mansion
    # currentRoom is the room the player is currently in (which can
    # be one of r1 through r4)
    # since it needs to be changed in the main part of the program,
    # it must be global
    global currentRoom

    # create the rooms and give them meaningful names
    r1 = Room("Room 1")
    r2 = Room("Room 2")
    r3 = Room("Room 3")
    r4 = Room("Room 4")

    # add exits to room 1
    r1.addExit("east", r2)   # -> to the east of room 1 is room 2
    r1.addExit("south", r3)
    # add grabbables to room 1
    r1.addGrabbable("key")
    # add items to room 1
    r1.addItem("chair", "It is made of wicker and no one is sitting
on it.")
    r1.addItem("table", "It is made of oak.  A golden key rests on
it.")

    # add exits to room 2
    r2.addExit("west", r1)
    r2.addExit("south", r4)
    # add items to room 2
    r2.addItem("rug", "It is nice and Indian.  It also needs to be
vacuumed.")
    r2.addItem("fireplace", "It is full of ashes.")

    # add exits to room 3
    r3.addExit("north", r1)
    r3.addExit("east", r4)
    # add grabbables to room 3
    r3.addGrabbable("book")
    # add items to room 3
    r3.addItem("bookshelves", "They are empty.  Go figure.")
    r3.addItem("statue", "There is nothing special about it.")
    r3.addItem("desk", "The statue is resting on it.  So is a book.")

    # add exits to room 4
    r4.addExit("north", r2)
    r4.addExit("west", r3)
    r4.addExit("south", None)     # DEATH!
    # add grabbables to room 4
    r4.addGrabbable("6-pack")
```

```
     # add items to room 4
     r4.addItem("brew_rig", "Gourd is brewing some sort of oatmeal
stout on the brew rig.  A 6-pack is resting beside it.")

     # set room 1 as the current room at the beginning of the game
     currentRoom = r1
```

Note the exit to the south of room 4: None. This is the "window" referred to earlier. If the player exits south in room 4, the game will be over. This will be easy to check since the current room will be None (and not some actual instance of the class room). Recall that the reserved word **None** in Python refers to nothing or the absence of a value.

When the program is first run, the player's inventory list is initially empty. Subsequently, the call to the `createRooms()` function creates the rooms and stores them in memory. In addition, the global variable *currentRoom* is set to the local variable *r1* (room 1). Even though *r1* is local to the function `createRooms()`, it will still be in memory for the duration of the game.

At this point, all that's left to do is to display the information associated with the current room, to prompt the player for an input action, and to act based on the player's input action. This will be done beneath the call to the function `createRooms()`. First, let's provide situational awareness and also deal with the possibility that the player has *jumped out the window* (in which case the game should end). These tasks will occur until either the player dies or asks to leave the game. Therefore, the remainder of the program will be contained within a repetition construct (we'll use a while loop). Exiting the while loop will be possible by use of the `break` instruction:

```
# play forever (well, at least until the player dies or asks to quit)
while (True):
     # set the status so the player has situational awareness
     # the status has room and inventory information
     status = "{}\nYou are carrying: {}\n".format(currentRoom,
inventory)

     # if the current room is None, then the player is dead
     # this only happens if the player goes south when in room 4
     # exit the game
     if (currentRoom == None):
          death()
          break

     # display the status
     print "============================================================"
     print status
```

Initially, a default status is set that provides the information associated with the current room and the player's inventory. This is then displayed. However, if the current room is None (i.e., the player exited south in room 4), then the player is dead and the game is ended (via the break statement that will exit the while loop). Note the call to the function `death()`. For now, you can comment this one out. It simply

displays an appropriate “message” when the player dies. It will be provided later in the full code listing for the main part of the game.

The next task is to add support for user input:

```
     # prompt for player input
     # the game supports a simple language of <verb> <noun>
     # valid verbs are go, look, and take
     # valid nouns depend on the verb
     # we use raw_input here to treat the input as a string instead of
     # an expression
     action = raw_input("What to do? ")

     # set the user's input to lowercase to make it easier to compare
     # the verb and noun to known values
     action = action.lower()

     # exit the game if the player wants to leave (supports quit,
     # exit, and bye)
     if (action == "quit" or action == "exit" or action == "bye"):
          break
```

The `raw_input` function is used instead of the familiar `input` function since it interprets the user's input as a string by default (unlike the `input` function). The strategy is to prompt the player for input and convert it to lowercase to make comparison to the supported vocabulary easier. We also handle exiting the game. The next task is to parse the player's action; that is, to determine the input and try to apply it to the supported vocabulary of a verb followed by a noun:

```
     # set a default response
     response = "I don't understand.  Try verb noun.  Valid verbs are
go, look, and take"
     # split the user input into words (words are separated by spaces)
     # and store the words in a list
     words = action.split()

     # the game only understands two word inputs
     if (len(words) == 2):
          # isolate the verb and noun
          verb = words[0]
          noun = words[1]
```

At this point, the user's input is parsed into a verb and a noun. The next step is to check the verb to see if it matches one that is supported (i.e., *go*, *look*, *take*):

```
          # the verb is: go
          if (verb == "go"):
               # set a default response
               response = "Invalid exit."

               # check for valid exits in the current room
```

```
                for i in range(len(currentRoom.exits)):
                     # a valid exit is found
                     if (noun == currentRoom.exits[i]):
                          # change the current room to the one that is
                          # associated with the specified exit
                          currentRoom = currentRoom.exitLocations[i]

                          # set the response (success)
                          response = "Room changed."

                          # no need to check any more exits
                          break
```

If the verb is *go*, we then check the noun for a valid exit in the current room. Recall that the exits have an associated exit location (via the parallel lists declared in the room class). The strategy is to identify the specified exit in the list of exits in the current room, and then identify the matching exit location (i.e., the room that the exit leads to). Subsequently, we change the current room to this adjacent room. We then break out of the for loop which will display the response (shown later) and cycle back to the beginning of the while loop.

Support for the verb *look* is similar:

```
          # the verb is: look
          elif (verb == "look"):
               # set a default response
               response = "I don't see that item."

               # check for valid items in the current room
               for i in range(len(currentRoom.items)):
                    # a valid item is found
                    if (noun == currentRoom.items[i]):
                         # set the response to the item's description
                         response = currentRoom.itemDescriptions[i]

                         # no need to check any more items
                         break
```

Support for the verb *take* is only slightly different:

```
          # the verb is: take
          elif (verb == "take"):
               # set a default response
               response = "I don't see that item."

               # check for valid grabbable items in the current room
               for grabbable in currentRoom.grabbables:
                    # a valid grabbable item is found
                    if (noun == grabbable):
                         # add the grabbable item to the player's
                         # inventory
```

```
                    inventory.append(grabbable)

                    # remove the grabbable item from the room
                    currentRoom.delGrabbable(grabbable)

                    # set the response (success)
                    response = "Item grabbed."

                    # no need to check any more grabbable items
                    break
```

In this case, if the specified grabbable item is found in the room, it is added to the player's inventory. Subsequently, it is removed from the current room's list of grabbables (after all, we don't want to grab it again!). An appropriate response is set, and the for loop is exited.

The last thing to do is to display the response and cycle back to the beginning of the while loop so that the description of the current room and the player's inventory can be displayed, and input can be solicited from the player once again:

```
     # display the response
     print "\n{}".format(response)
```

Yes, this is a lot of code split up into many parts across this document. This is why it was suggested that you not actually write any segmented code as you read this document and process through the activity. And now, here is the entire source code for the main part of the program. Note that the `createRooms()` function is not included below; however, it should be placed above the main part of the program (but outside of the `Room` class):

```
################################################################
# START THE GAME!!!
inventory = [] # nothing in inventory...yet
createRooms()  # add the rooms to the game

# play forever (well, at least until the player dies or asks to quit)
while (True):
     # set the status so the player has situational awareness
     # the status has room and inventory information
     status = "{}\nYou are carrying: {}\n".format(currentRoom,
inventory)

     # if the current room is None, then the player is dead
     # this only happens if the player goes south when in room 4
     if (currentRoom == None):
          status = "You are dead."

     # display the status
     print "========================================================"
     print status
```

```
     # if the current room is None (and the player is dead), exit the
     # game
     if (currentRoom == None):
          death()
          break

     # prompt for player input
     # the game supports a simple language of <verb> <noun>
     # valid verbs are go, look, and take
     # valid nouns depend on the verb
     # we use raw_input here to treat the input as a string instead of
     # a numeric value
     action = raw_input("What to do? ")

     # set the user's input to lowercase to make it easier to compare
     # the verb and noun to known values
     action = action.lower()

     # exit the game if the player wants to leave (supports quit,
     # exit, and bye)
     if (action == "quit" or action == "exit" or action == "bye"):
          break

     # set a default response
     response = "I don't understand.  Try verb noun.  Valid verbs are
go, look, and take"
     # split the user input into words (words are separated by spaces)
     words = action.split()

     # the game only understands two word inputs
     if (len(words) == 2):
          # isolate the verb and noun
          verb = words[0]
          noun = words[1]

          # the verb is: go
          if (verb == "go"):
               # set a default response
               response = "Invalid exit."

               # check for valid exits in the current room
               for i in range(len(currentRoom.exits)):
                    # a valid exit is found
                    if (noun == currentRoom.exits[i]):
                         # change the current room to the one that is
                         # associated with the specified exit
                         currentRoom = currentRoom.exitLocations[i]
```

```
                    # set the response (success)
                    response = "Room changed."

                    # no need to check any more exits
                    break
          # the verb is: look
          elif (verb == "look"):
               # set a default response
               response = "I don't see that item."

               # check for valid items in the current room
               for i in range(len(currentRoom.items)):
                    # a valid item is found
                    if (noun == currentRoom.items[i]):
                         # set the response to the item's description
                         response = currentRoom.itemDescriptions[i]

                         # no need to check any more items
                         break
          # the verb is: take
          elif (verb == "take"):
               # set a default response
               response = "I don't see that item."

               # check for valid grabbable items in the current room
               for grabbable in currentRoom.grabbables:
                    # a valid grabbable item is found
                    if (noun == grabbable):
                         # add the grabbable item to the player's
                         # inventory
                         inventory.append(grabbable)

                         # remove the grabbable item from the room
                         currentRoom.delGrabbable(grabbable)

                         # set the response (success)
                         response = "Item grabbed."

                         # no need to check any more grabbable items
                         break

     # display the response
     print "\n{}".format(response)
```

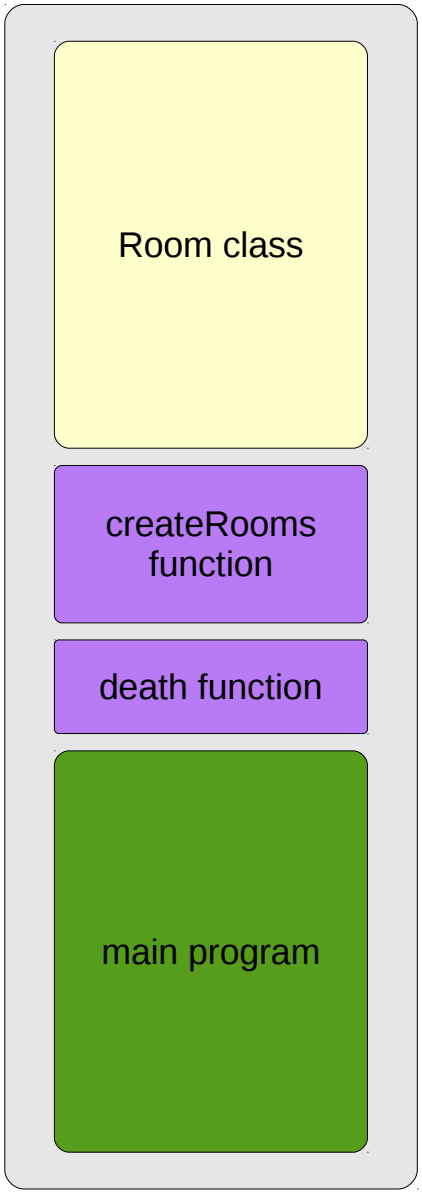
Source code template?

You may be asking yourself if there is source code in the form of a template for this activity. No, there isn't. Although it will be tedious and a bit time-consuming, typing the code by hand will help you to

understand the code, to learn how to properly format Python code, and perhaps result in better retention of Python syntax and good program style and structure. Please refrain from simply copy-and-pasting the source code from the PDF version of this document into a Python source code file. It is to your advantage to take time now to learn this so that it becomes easier in the future. After all, aren't you in this curriculum because you are interested in learning this stuff and because you chose to be here? In fact, the prof should code the game with you in real time during this activity, typing each statement, one at a time!

The flow of the source code is as follows:

- The room class;
- The `createRooms()` function;
- The (optional) `death()` function; and
- The main part of the program.



The optional `death()` function

Earlier, you were asked to comment out the call to the function `death()` in the main part of the program. In case you wish to implement it, here it is (yes, it is intentionally obfuscated). It must be defined above the main part of the program. Since it is obfuscated, you may copy-and-paste this function into your source code (pay attention to the end of the lines and indentation!):

```
# displays an appropriate "message" when the player dies
# yes, this is intentionally obfuscated!
def death():
     print " " * 17 + "u" * 7
     print " " * 13 + "u" * 2 + "$" * 11 + "u" * 2
     print " " * 10 + "u" * 2 + "$" * 17 + "u" * 2
     print " " * 9 + "u" + "$" * 21 + "u"
     print " " * 8 + "u" + "$" * 23 + "u"
     print " " * 7 + "u" + "$" * 25 + "u"
     print " " * 7 + "u" + "$" * 25 + "u"
     print " " * 7 + "u" + "$" * 6 + "\"" + " " * 3 + "\"" + "$" * 3 +
"\"" + " " * 3 + "\"" + "$" * 6 + "u"
     print " " * 7 + "\"" + "$" * 4 + "\"" + " " * 6 + "u$u" + " " * 7
+ "$" * 4 + "\""
     print " " * 8 + "$" * 3 + "u" + " " * 7 + "u$u" + " " * 7 + "u" +
"$" * 3
     print " " * 8 + "$" * 3 + "u" + " " * 6 + "u" + "$" * 3 + "u" + "
" * 6 + "u" + "$" * 3
     print " " * 9 + "\"" + "$" * 4 + "u" * 2 + "$" * 3 + " " * 3 +
"$" * 3 + "u" * 2 + "$" * 4 + "\""
     print " " * 10 + "\"" + "$" * 7 + "\"" + " " * 3 + "\"" + "$" * 7
+ "\""
     print " " * 12 + "u" + "$" * 7 + "u" + "$" * 7 + "u"
     print " " * 13 + "u$\"$\"$\"$\"$\"$\"$u"
     print " " * 2 + "u" * 3 + " " * 8 + "$" * 2 + "u$ $ $ $ $u" + "$"
* 2 + " " * 7 + "u" * 3
     print " u" + "$" * 4 + " " * 8 + "$" * 5 + "u$u$u" + "$" * 3 + "
" * 7 + "u" + "$" * 4
     print " " * 2 + "$" * 5 + "u" * 2 + " " * 6 + "\"" + "$" * 9 +
"\"" + " " * 5 + "u" * 2 + "$" * 6
     print "u" + "$" * 11 + "u" * 2 + " " * 4 + "\"" * 5 + " " * 4 +
"u" * 4 + "$" * 10
     print "$" * 4 + "\"" * 3 + "$" * 10 + "u" * 3 + " " * 3 + "u" * 2
+ "$" * 9 + "\"" * 3 + "$" * 3 + "\""
     print " " + "\"" * 3 + " " * 6 + "\"" * 2 + "$" * 11 + "u" * 2 +
" " + "\"" * 2 + "$" + "\"" * 3
     print " " * 11 + "u" * 4 + " \"\"" + "$" * 10 + "u" * 3
     print " " * 2 + "u" + "$" * 3 + "u" * 3 + "$" * 9 + "u" * 2 +
" \"\"" + "$" * 11 + "u" * 3 + "$" * 3
     print " " * 2 + "$" * 10 + "\"" * 4 + " " * 11 + "\"\"" + "$" *
11 + "\""
```

```
    print " " * 3 + "\"" + "$" * 5 + "\"" + " " * 22 + "\"\"" + "$" *
4 + "\"\""
    print " " * 5 + "$" * 3 + "\"" + " " * 25 + "$" * 4 + "\""
```

Suggested improvements

Although this is a simple game in terms of gameplay, it can quickly become quite complicated to design and implement. It can also become more involved and, dare I say it, even more fun than it already is by making just a few fairly minor improvements:

- Some items that can be placed in the player's inventory are identified in the description of items (e.g., the key on the table in room 1). When placed in inventory, the item's description should be appropriately modified (e.g., the key is no longer on the table).
- Adding more observable items and items that can be added to inventory in each room.
- Adding more rooms (e.g., making some sort of maze).
- Making the mansion three-dimensional (i.e., adding exits above and below to new rooms).
- Adding a goal that requires solving a puzzle. For example, using a key in inventory that was taken from a room to unlock a box that is located in another room. Unlocking the box may reveal new items. This would require adding to the vocabulary (e.g., a new verb *use*). If the player is in the correct room and has the correct item in inventory, issuing the proper action (e.g., "use key") would solve a puzzle and/or reveal new items that can be observed or taken into inventory.
- Adding the ability to look at individual inventory items (i.e., inventory items can have descriptions as well).
- Adding points that the player can accumulate by adding items to inventory or by solving puzzles.
- ...there are more...

Homework: Room Adventure

For the homework portion of this activity, you may have the option to work in **groups** (pending prof approval). It is suggested that groups contain at least one confident Python coder.

Your task is to implement **at least one** of the suggested improvements. **Clearly comment your changes and additions to the source code!** If you wish to implement an improvement that is not listed above, please get with the prof first for approval. Note that this component is worth **over one-third of your grade** for this assignment! So be creative and precise. You will definitely want to implement significant improvements, since your grade for this portion will be comparatively assigned (i.e., the best improvements to the game earn the best grade for this portion of the assignment).

You are to submit your Python source code only (as a .py file) through the upload facility on the web site.

Raspberry Pi Activity: My Binary Addiction

In this activity, you will implement a one-bit binary adder using LEDs, resistors, and push-button switches. You will need the following items:

- Raspberry Pi B v3 with power adapter;
- LCD touchscreen;
- Keyboard and mouse;
- Breadboard;
- GPIO interface board with ribbon cable; and
- LEDs, resistors, switches, and jumper wires provided in your kit.

Regarding the electronic components, you will need the following:

- 2x red LEDs;
- 4x green LEDs;
- 4x blue LEDs;
- 2x push-button switches;
- 9x 220Ω resistors; and
- 9x jumper wires.

The adder

Recall the single-bit half adder shown in a previous lesson:

It takes two single-bit inputs, A and B, and produces two outputs, S (the sum bit) and C (the carry bit). The half adder has the following truth table:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The question is, how do we implement these logic gates in a programming language? We have seen that if-statements are related to logic gates. For example, we can evaluate if one condition *and* another are true. If and only if both are true will the entire condition be true (and the statements in the true part of the if-statement will be executed). Therefore, one way to implement the truth table for a half adder is as follows:

```
 1: if A is 0 and B is 0
 2: then
 3:     S ← 0
 4:     C ← 0
 5: else
 6:     if A is 1 and B is 1
 7:     then
 8:         S ← 0
 9:         C ← 1
10:     else
11:         S ← 1
12:         C ← 0
13:     end
14: end
```

Notice that this basically handles each row of the truth table above. The first if-statement handles the first row of the truth table (when A and B are both 0), and sets S and C to 0. The second if-statement handles the last row of the truth table (when A and B are both 1), and sets S to 0 and C to 1. The last case (the else part of the second if-statement) handles the two middle rows of the truth table, where either A or B is 1 (but not both), and sets S to 1 and C to 0.

This is how we would implement a half adder in Scratch, for example. Most general purpose programming languages (like Python), however, allow bitwise operations. That is, they can take Boolean inputs (like A and B) and implement the logic of primitive gates (e.g., *and* and *or*). This is a much simpler way to implement the logic! Plus, it allows us to significantly reduce the amount of code required to implement the half adder. Recall that S is the output of A *xor* B, and C is the output of A *and* B:

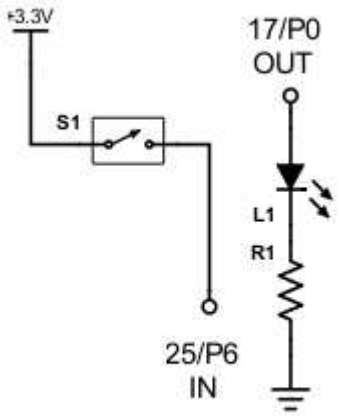
```
S = (~A & B) | (A & ~B)
C = A & B
```

That's it! Just two statements. Recall that Python support several bitwise operators, including *and* (&), *or* (|), and *not* (~). Also, the *xor* operation is performed as shown in an earlier lesson: *not* A *and* B *or* A *and not* B. Therefore, S is ultimately assigned the result of A *xor* B, and C is assigned the result of A *and* B. Although we structured our half adder such that the *xor* functionality was built using the three primitive gates (*and*, *or*, and *not*), Python has the *xor* bitwise operator (^) that we can use directly! This is quite useful:

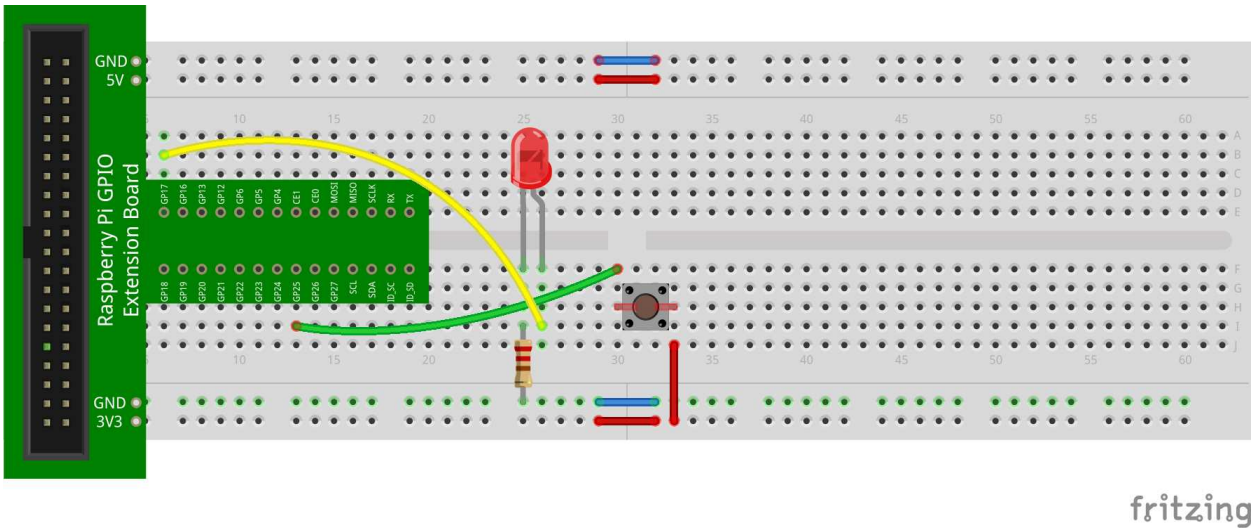
```
S = A ^ B
C = A & B
```

GPIO in Python

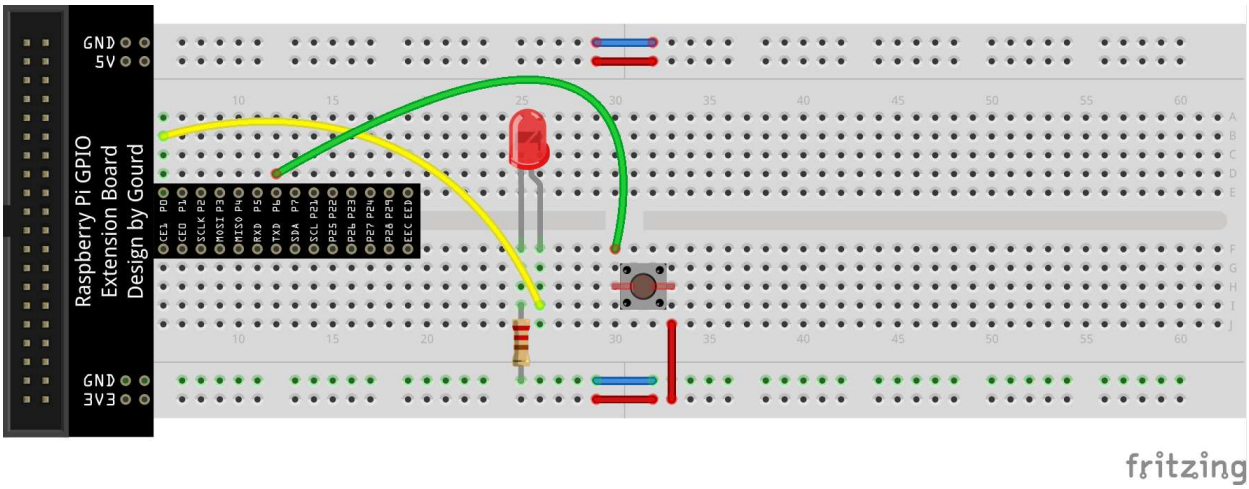
Before we continue, let's review how Python handles GPIO on the RPi. Implement the following single switch, single LED circuit:



Here's one way to layout this circuit:



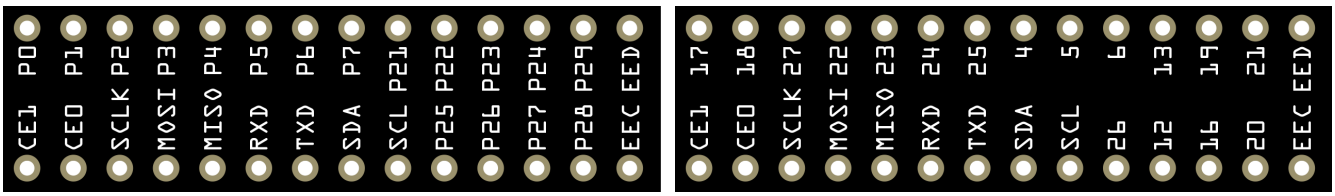
If you have the black GPIO interface board, layout the circuit as follows instead:



Recall that there are actually **three** different pin numbering schemes in use with GPIO pins on the RPi: (1) the **physical** pin order on the RPi; (2) the numbering assigned by the manufacturer of the **Broadcom** chip on the RPi; and (3) an older numbering assigned by an early RPi user who developed a library called **wiringPi**. Here's the cross-reference table shown in an earlier activity:

BCM	wPi	Name	Physical		Name	wPi	BCM
		3V3	1	2	5V		
2	8	SDA.1	3	4	5V		
3	9	SCL.1	5	6	GND		
4	7	GPIO.7	7	8	TXD	15	14
		GND	9	10	RXD	16	15
17	0	GPIO.0	11	12	GPIO.1	1	18
27	2	GPIO.2	13	14	GND		
22	3	GPIO.3	15	16	GPIO.4	4	23
		3V3	17	18	GPIO.5	5	24
10	12	MOSI	19	20	GND		
9	13	MISO	21	22	GPIO.6	6	25
11	14	SCLK	23	24	CE0	10	8
		GND	25	26	CE1	11	7
0	30	SDA.0	27	28	SCL.0	31	1
5	21	GPIO.21	29	30	GND		
6	22	GPIO.22	31	32	GPIO.26	26	12
13	23	GPIO.23	33	34	GND		
19	24	GPIO.24	35	36	GPIO.27	27	16
26	25	GPIO.25	37	38	GPIO.28	28	20
		GND	39	40	GPIO.29	29	21

If you have the green GPIO interface, you won't have to refer to the table since the RPi uses the BCM pin numbering scheme (which the green GPIO interface also uses). If you have the black GPIO interface, the following comparison of the GPIO interface boards labeled with both pin numbering schemes (shown in an earlier activity) will help:



In the layout diagram above, the LED is connected to **GP17** (which refers to BCM pin **17** on the RPi and **P0** on the black GPIO interface), and the switch is connected to **GP25** (which refers to BCM pin **25** on the RPi and **P6** on the black GPIO interface).

The goal is to detect a switch press by ensuring that the input pin to which it is connected is initially pulled down. When the switch is pressed, current flows from +3.3V to the input pin, which can be detected. The LED is then driven high. Here's a Python program that implements this:

```
import RPi.GPIO as GPIO
from time import sleep

# set the LED and switch pin numbers
led = 17
```

```
button = 25

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the LED and switch pins
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

# do this forever
while (True):
     # light the LED when the switch is pressed...
     # ...turn it off otherwise
     if (GPIO.input(button) == GPIO.HIGH):
          GPIO.output(led, GPIO.HIGH)
     else:
          GPIO.output(led, GPIO.LOW)
     sleep(0.1)
```

To make things more interesting, let's blink an LED once every second (i.e., on for 0.5 second, off for 0.5 second) by default. If the switch is pressed, let's blink the LED faster, once every 0.5 second (i.e., on for 0.25 second, off for 0.25 second):

```
import RPi.GPIO as GPIO
from time import sleep

# set the LED and switch pin numbers
led = 17
button = 25

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the LED and switch pins
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# we'll discuss this later, but the try-except construct allows
# us to detect when Ctrl+C is pressed so that we can reset the
# GPIO pins
try:
     # blink the LED forever
     while (True):
          # the delay is 0.5s if the switch is not pressed
          if (GPIO.input(button) == GPIO.HIGH):
               delay = 0.5
          # otherwise, it's 0.25s
          else:
               delay = 0.25
```

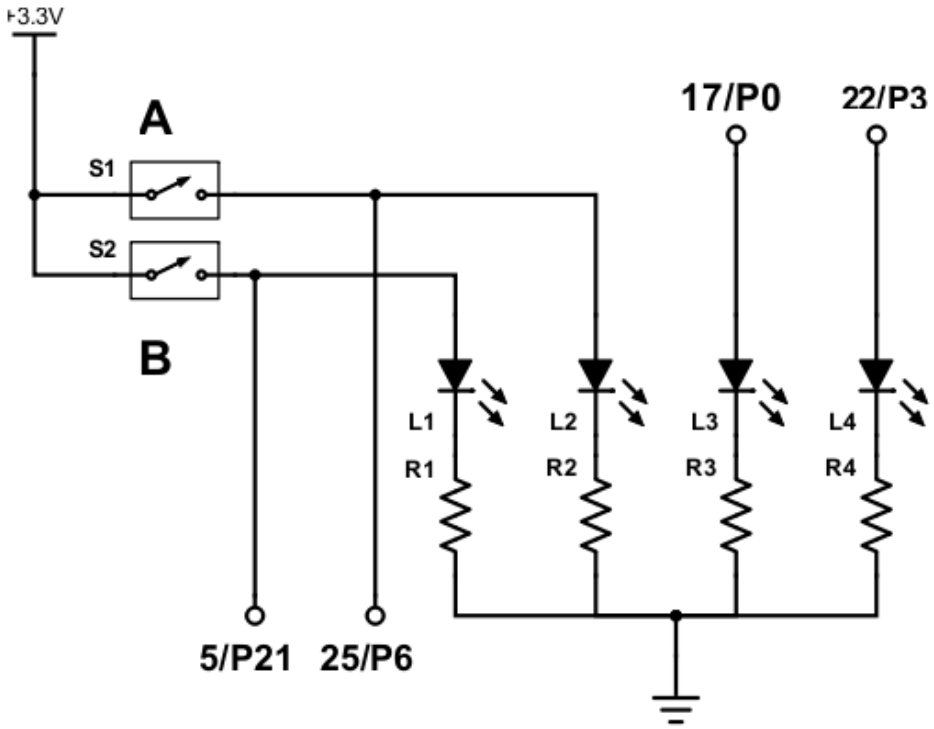
```
            # blink the LED
            GPIO.output(led, GPIO.HIGH)
            sleep(delay)
            GPIO.output(led, GPIO.LOW)
            sleep(delay)
# detect Ctrl+C
except KeyboardInterrupt:
     # reset the GPIO pins
     GPIO.cleanup()
```

You probably noticed the **try-except** construct. A comment notes that it will be discussed later (and it will!). For now, it's enough to know that such a construct is used to group statements that may cause an exception (i.e., some sort of abnormal event during runtime). In the case of the program above, the abnormal event is the user pressing Ctrl+C (which breaks out of the program). We can detect this and execute statements subsequently to do things like cleaning up and/or resetting the GPIO pins.

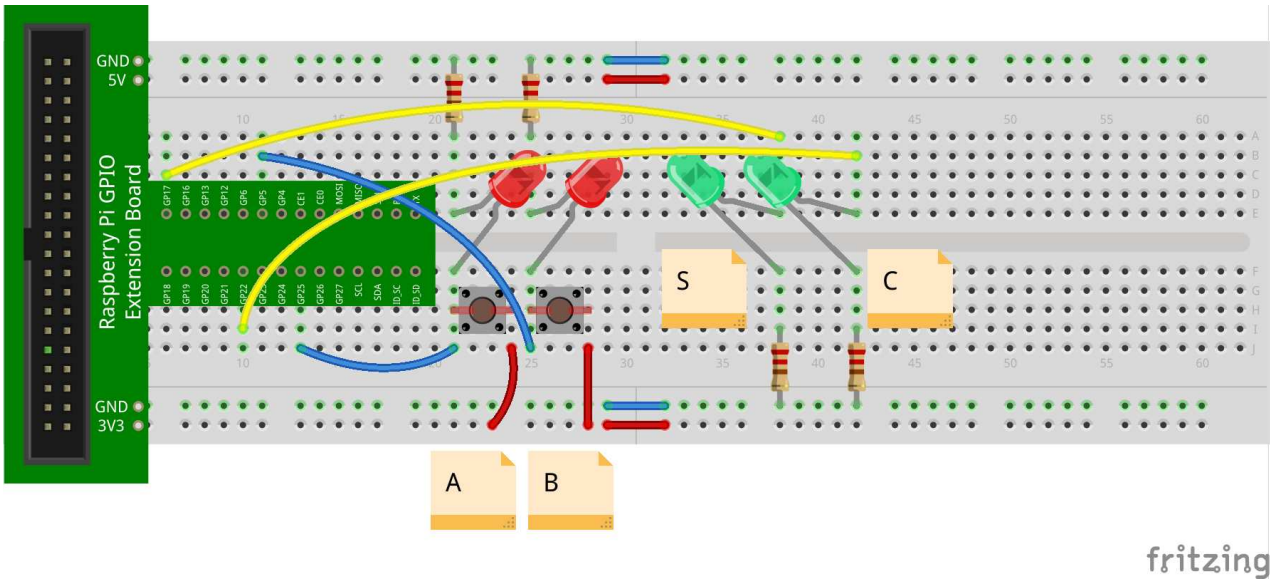
Now that GPIO on the RPi in Python has been reviewed, let's get to work on the circuit for this activity.

The circuit

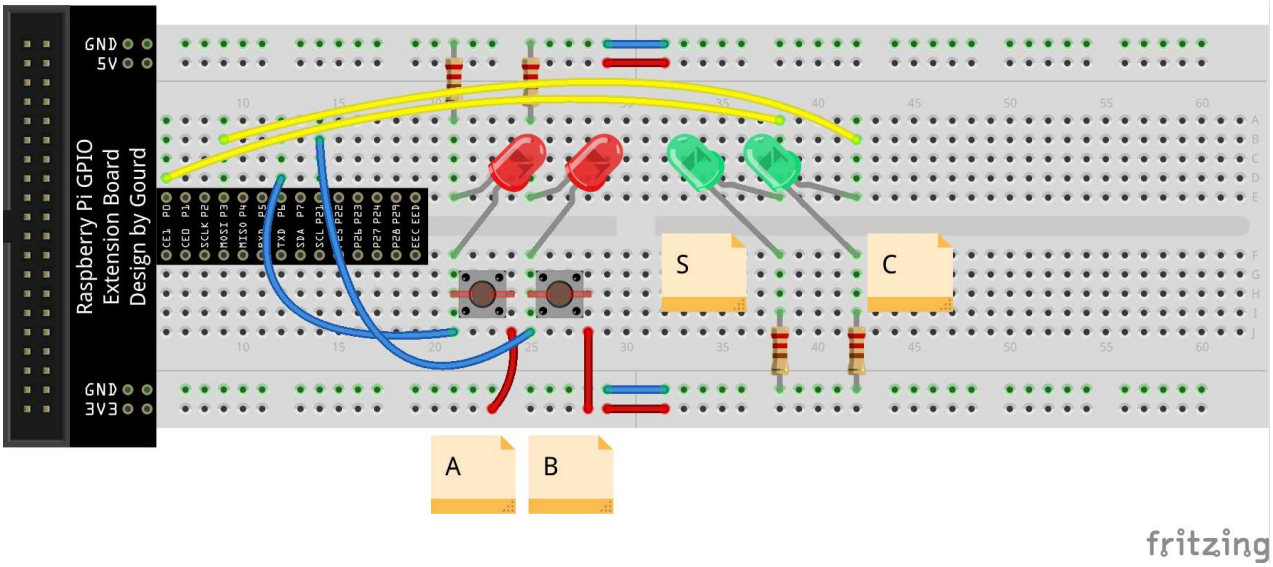
Implement the following half adder circuit. For this part of the activity, you will need two switches, four LEDs, four resistors, and some jumper wires:



Here's one way to layout this circuit:



If you have the black GPIO interface, layout the circuit as follows instead:



Note that the LEDs labeled S and C (the outputs) are green LEDs, and the LEDs labeled A and B (the inputs) are red LEDs. The input LEDs are also connected to the push-button switches. Since the switches are connected to +3.3V (i.e., they complete the circuit both to the input pins, GP25 (P6) and GP5 (P21), and to the red LEDs when closed), then the LEDs must be connected such that the anode (the longer positive side) is matched with the switch (i.e., the shorter negative side is connected to GND). This is illustrated in the circuit above. Pay close attention to polarity (i.e., where the negative and positive terminals of electronic components are) and wiring. Recall that the position of the resistor (either on the negative or positive side of the LED) doesn't matter. In the circuit above, the resistors are placed on the negative side of the LEDs.

Note that S (the green LED on the left) is connected to GP17 (P0), and C (the green LED on the right) is connected to GP22 (P3).

Note that the labels (A, B, S, and C) are strictly informative (i.e., they serve no function other than to provide situational awareness). It should be clear that the left push-button switch represents the bit input A, the right push-button switch represents the bit input B, the green LED on the left represents the bit output S, and the green LED on the right represents the bit output C. The red LEDs are wired to the push-button switches and provide feedback of the state of A and B (i.e., the left LED corresponds to the left push-button switch, and vice versa).

The code

Here's the entire program for the half adder in Python:

```
import RPi.GPIO as GPIO
from time import sleep

# set the GPIO pin numbers
inA = 25
inB = 5
outS = 17
outC = 22

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the input and output pins
GPIO.setup(inA, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(inB, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(outS, GPIO.OUT)
GPIO.setup(outC, GPIO.OUT)

# we'll discuss this later, but the try-except construct allows
# us to detect when Ctrl+C is pressed so that we can reset the
# GPIO pins
try:
     # keep going until the user presses Ctrl+C
     while (True):
          # initialize A, B, S, and C
          A = 0
          B = 0
          S = 0
          C = 0

          # set A and B depending on the switches
          if (GPIO.input(inA) == GPIO.HIGH):
               A = 1
          if (GPIO.input(inB) == GPIO.HIGH):
               B = 1

          # calculate S and C using A and B
          S = A ^ B      # A xor B
```

```
            C = A & B      # A and B

            # set the output pins appropriately
            # (to light the LEDs as appropriate)
            GPIO.output(outS, S)
            GPIO.output(outC, C)
    # detect Ctrl+C
    except KeyboardInterrupt:
         # reset the GPIO pins
         GPIO.cleanup()
```

After importing the required libraries, variables that map to GPIO pins are declared (*inA* representing the pin connected to switch A, *inB* representing the pin connected to switch B, *outS* representing the pin connected to LED S, and *outC* representing the pin connected to LED C). Next, the pins are setup (as either input or output pins). Since the switches are wired to +3.3V, the input pins are setup with a pull-down resistor. That is, their default value will be low at 0V. When a switch is pressed, this will bring up the input pin to 3.3V. Since the LEDs representing the inputs are also wired with the switches, they will light when the switches are pressed.

Next, the program detects the switch states and turns on the LEDs as appropriate. A and B are initialized to 0. If a switch is pressed, its corresponding input (A or B) is changed to 1. The values of S and C are then calculated (as A *xor* B for S, and A *and* B for C). Finally, the LEDs are triggered appropriately depending on the values of S and C.

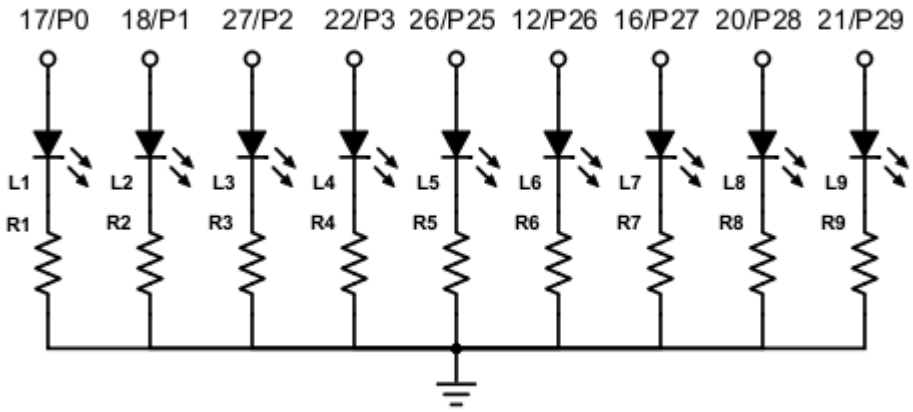
Perhaps a more efficient way to assign values for A and B is simply to modify the while loop as follows:

```
    while (True):
         # set A and B depending on the switches
         A = GPIO.input(inA)
         B = GPIO.input(inB)

         # calculate S and C depending on A and B
         S = A ^ B # A xor B
         C = A & B # A and B

         # set the output pins appropriately (to light the LEDs as
         #  appropriate)
         GPIO.output(outS, S)
         GPIO.output(outC, C)
```

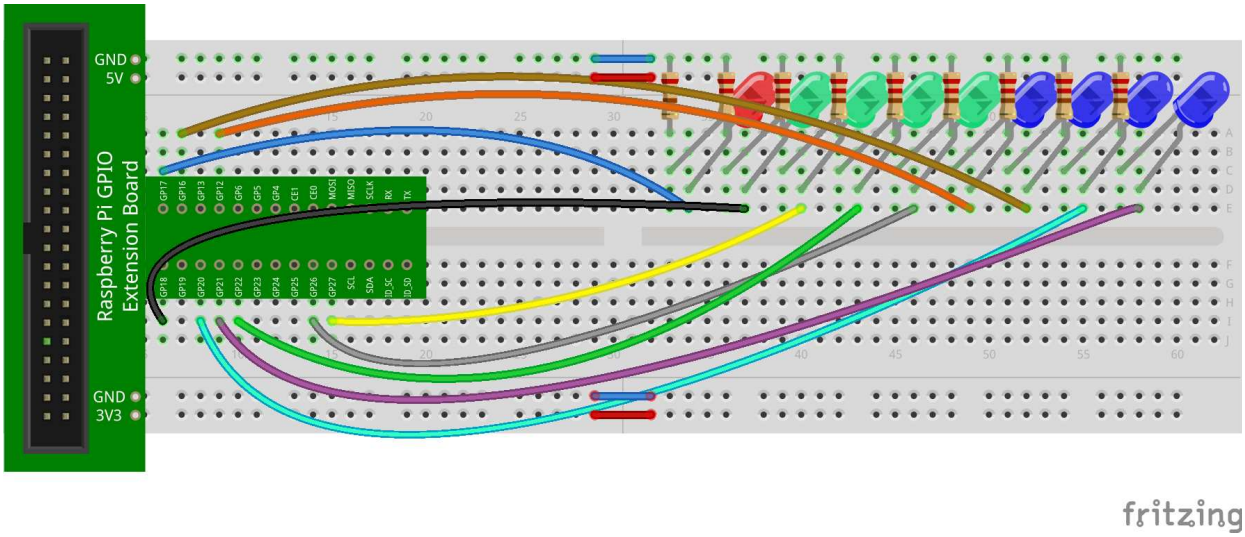
The end result is the same!

Extending this a bit

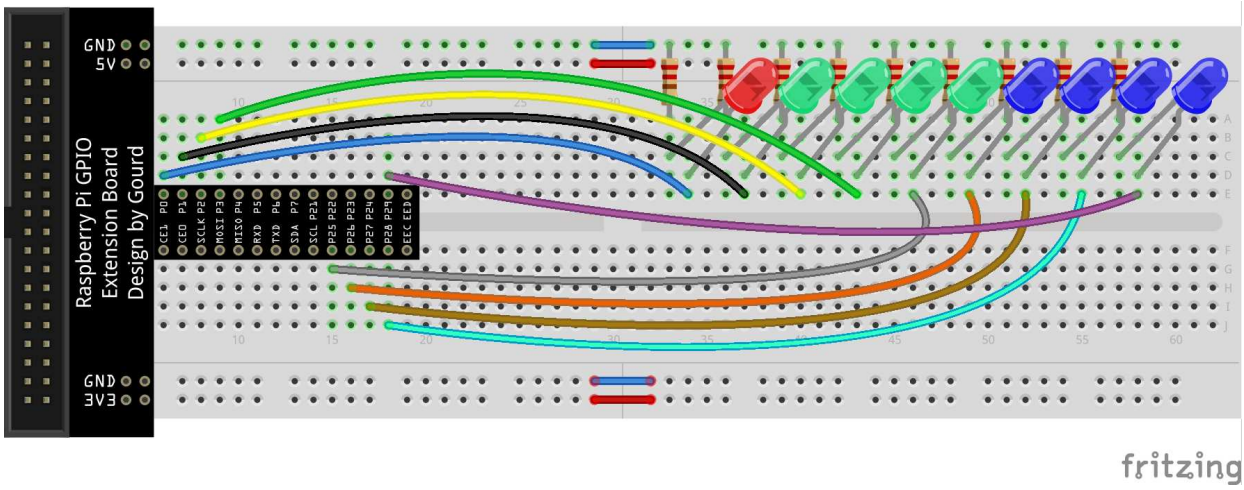
Take a look at the following circuit diagram. This time, there are nine LEDs, all connected to GPIO pins (GP17=P0, GP18=P1, GP27=P2, GP22=P3, GP26=P25, GP12=P26, GP16=P27, GP20=P28, GP21=P29) to a resistor that is connected to ground.



Here is one way to layout this circuit:



For the black GPIO interface, the layout diagram could look like this:



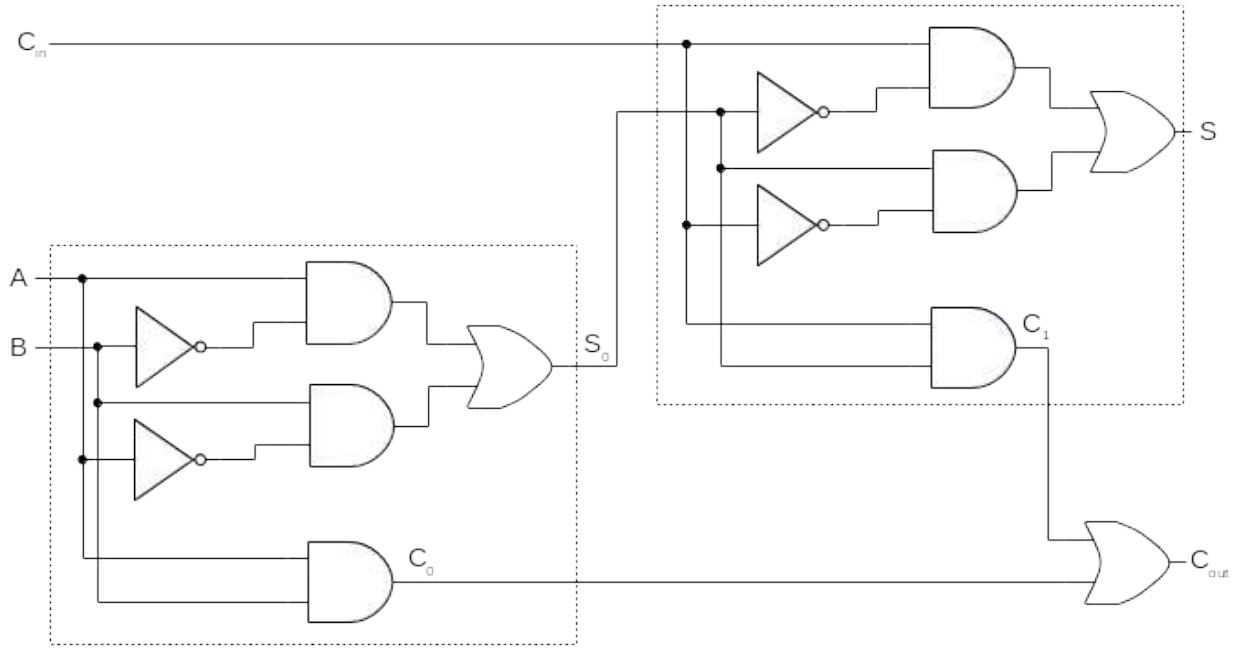
The LEDs represent the sum of two 8-bit numbers, with the least significant bit represented by the LED all the way to the right. For example, if the LEDs were to represent the sum of 94 + 113 = 207 (see the table below), then the state of the LEDs would be: off, on, on, off, off, on, on, on, on. The overflow bit (on the left) would be 0 (off).

	Binary										Decimal
	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Carry		0	1	1	1	0	0	0	0		1
1st number			0	1	0	1	1	1	1	0	94
2nd number			0	1	1	1	0	0	0	1	113
Sum		0	1	1	0	0	1	1	1	1	207

If the LEDs were to represent the sum of 150 + 150 = 300 (see the table below), then the state of the LEDs would be: on, off, off, on, off, on, on, off, off. In this case, the overflow bit would be 1 (on).

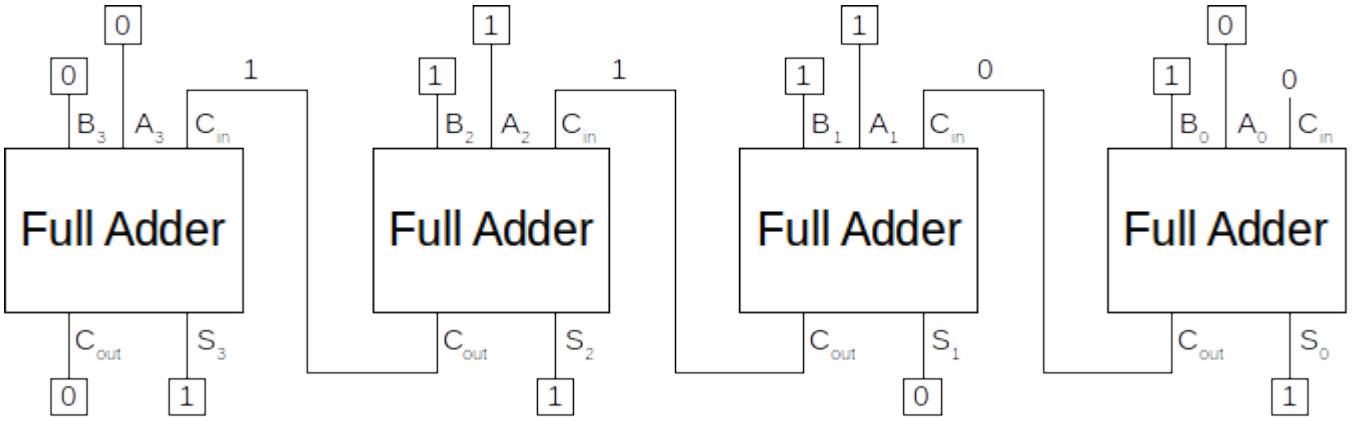
	Binary										Decimal
	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Carry		1	0	0	1	0	1	1	0		1
1st number			1	0	0	1	0	1	1	0	150
2nd number			1	0	0	1	0	1	1	0	150
Sum		1	0	0	1	0	1	1	0	0	300

To make this work, you will need to implement a full adder as described in a previous lesson:



Recall that a full adder is made up of two half adders. One half adder computes the sum and carry of A and B. The sum is then brought into another half adder and added along with the carry in. The sum of this second half adder produces the actual sum of A and B plus the carry in. The carry out of this half adder is combined with the carry out of the first half adder through an *or* gate. The output is the carry out. You can take the script that implements the half adder (created in the first part of this activity) and extend it to a full adder.

Since each number is represented as a list, we will iterate through each, one bit at a time, and implement the full adder to produce a sum and carry out for each bit. Recall that the carry out is fed into the carry in for the next bit (to the left). We saw this when chaining full adders together to add two 4-bit numbers together:



In this activity, we are extending this to add two 8-bit numbers. The idea is the same.

Let's take a look at the beginning of the source code for this program. First, the header:

```
##################################################################
# Name:
# Date:
# Description:
##################################################################
import RPi.GPIO as GPIO        # bring in GPIO functionality
from random import randint     # to generate random integers
```

We'll make use of the `randint` function from the **random** library to generate the two random numbers.

Since there are many outputs, one for each LED, why don't we specify them all in a list. the following function sets up the GPIO pins for this program:

```
# function that defines the GPIO pins for the nine output LEDs
def setGPIO():
    # define the pins (change these if they are different)
    gpio = [17, 18, 27, 22, 26, 12, 16, 20, 21]
    # set them up as output pins
    GPIO.setup(gpio, GPIO.OUT)

    return gpio
```

Note how the pins are defined in a list called *gpio*. We then set each pin in the list to be an output pin.

The following snippet of code defines a function that generate a random 8-bit binary number:

```
# function that randomly generates an 8-bit binary number
def setNum():
    # create an empty list to represent the bits
    num = []
    # generate eight random bits
    for i in range(0, 8):
        # append a random bit (0 or 1) to the end of the list
        num.append(randint(0, 1))

    return num
```

This function first creates an empty list, called *num*. It then appends a random integer from 0 to 1, 8 times. The final number is then returned.

The following function handles turning on the appropriate LEDs representing the sum of the two 8-bit binary numbers:

```
# displays the sum (by turning on the appropriate LEDs)
def display():
    for i in range(len(sum)):
        # if the i-th bit is 1, then turn the i-th LED on
        if (sum[i] == 1):
            GPIO.output(gpio[i], GPIO.HIGH)
        # otherwise, turn it off
        else:
            GPIO.output(gpio[i], GPIO.LOW)
```

The function first iterates through the bits in the final sum. For each bit that is on (1), the matching GPIO pin is set high. For each bit that is off (0), the matching GPIO pin is set low.

And now we have reached the function that you are to implement in this activity – the full adder:

```
# function that implements a full adder using two half adders
# inputs are Cin, A, and B; outputs are S and Cout
# this is the function that you need to implement
def fullAdder(Cin, A, B):
    ##########################
    # write your code here!!!!!
    ##########################

    return S, Cout      # we can return more than one value!
```

Of course, you will need to implement this on your own! At then end of the function, two values are returned: S and Cout. This makes perfect sense, since that's the expected output of a full adder.

The following function controls the addition of each bit in the two 8-bit binary numbers. It effectively serves as the chain that connects the full adders (that you will implement in the function above):

```
# controls the addition of each 8-bit number to produce a sum
def calculate(num1, num2):
    Cout = 0            # the initial Cout is 0
    sum = []            # initialize the sum
    n = len(num1) - 1   # position of the right-most bit of num1

    # step through each bit, from right-to-left
    while (n >= 0):
        # isolate A and B (the current bits of num1 and num2)
        A = num1[n]
        B = num2[n]
        # set the Cin (as the previous half adder's Cout)
        Cin = Cout

        # call the fullAdder function that takes Cin, A, and...
        # ...B, and returns S and Cout
        S, Cout = fullAdder(Cin, A, B)

        # insert sum bit, S, at the beginning (index 0) of sum
        sum.insert(0, S)

        # go to the next bit position (to the left)
        n -= 1

    # insert the final carry out at the beginning of the sum
    sum.insert(0, Cout)

    return sum
```

Once all of the bits have been run through the full adder (and the sum has been completely calculated), the overflow bit of the sum (i.e., the left-most bit at the first position in the list *sum*) is set as the final C_{out}. This is why the circuit requires nine LEDs.

And now we have reached the main part of the program:

```
# use the Broadcom pin scheme
GPIO.setmode(GPIO.BCM)

# setup the GPIO pins
gpio = setGPIO()

# get a random num1 and display it to the console
num1 = setNum()
print "     ", num1

# get a random num2 and display it to the console
num2 = setNum()
```

```
print "+    ", num2

# calculate the sum of num1 + num2 and display it to the console
sum = calculate(num1, num2)
print "= ", sum

# turn on the appropriate LEDs to "display" the sum
display()

# wait for user input before cleaning up and resetting GPIO pins
raw_input("Press ENTER to terminate")
GPIO.cleanup()
```

The main part of the program first sets the GPIO output pins (connected to the LEDs) by calling the function `setGPIO`. Again, this function defines a list that contains the pins corresponding to the nine output LEDs. It then iterates over them (via a for loop) and sets them up as output pins. The list is then returned to the main part of the program (note that the variable `gpio` contains this list).

Next, the first number is generated by calling the function `setNum`. Again, this function iteratively builds a list of eight random bits. Once finished, the list is returned to the main part of the program (note that the variable `num1` contains this list). The same occurs for the second number. Note the use of the `randint` function. It is imported through the *random* library. Its format is `randint(x, y)`, where *x* and *y* are the lower and upper values specified by the interval [*x*, *y*] to select a random integer from. For example, `randint(5, 44)` selects a random integer from 5 to 44.

Next, the sum is calculated by calling the function `calculate` (passing in the two numbers as parameters). Again, this function serves as the 8-bit adder that chains together eight full adders. It cycles through the two numbers from right-to-left, each time (i.e., for each bit) calling the `fullAdder` function. This function is provided C_{in}, A, and B as input parameters. It implements a full adder (made up of two half adders) and calculates (and returns) values for S and C_{out}. **Your task is to implement the `fullAdder` function.**

Finally, the `display` function is called, which turns on the appropriate LEDs that correspond to the bits that are on (1) in the variable `sum`.

Note the `raw_input` function near the end of the program. You have previously seen the `input` function (which allows user input to be provided and stored to a variable). The function `raw_input` is similar; however, it treats any input as a string. The regular `input` function attempts to evaluate the user input (which, for example, could be an integer). The function `raw_input` works in Python 2.7.x; however, it has been removed in Python 3.x (which only has the `input` function).

Homework: Full Adder...Reloaded

Create the *fullAdder* function that implements a full adder (that is made up of two half adders). Of course, you will also need to implement the program previously covered to test appropriately. **Make sure to use bitwise operators in your implementation of the full adder! A template is provided on the class web site.**

You are to submit your Python source code only (as a .py file) through the upload facility on the web site.

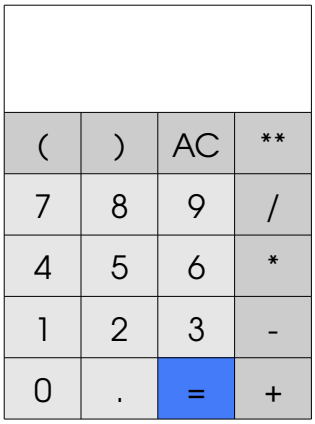
Raspberry Pi Activity: The Reckoner

In this activity, you will implement a simple graphical calculator using the Tkinter library. You will need the following items:

- Raspberry Pi B v3 with power adapter;
- LCD touchscreen; and
- Keyboard and mouse.

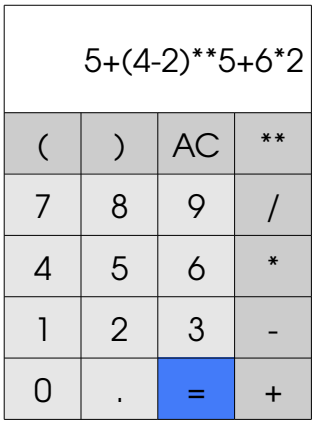
If you wish, you can simply bring your laptop with the Python interpreter (and also perhaps IDLE) installed since you will not be using the GPIO pins on the RPi. Note, however, that the calculator is specifically designed to illustrate an intuitive user interface for the LCD touchscreen included in your RPi kit. Therefore, it is actually to your advantage to do the activity on the RPi and LCD touchscreen.

Let's begin by taking a look at a model of the GUI for The Reckoner:

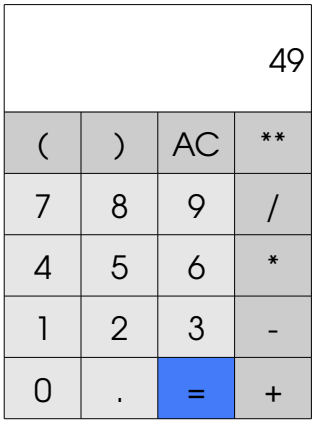


The top portion represents the display (i.e., where expressions and their results can be displayed). The bottom portion includes various buttons that either represent operands (by combining the digits 0 through 9 and/or the decimal point), operators (that perform various arithmetic operations), or special-purpose actions: clearing the display (AC) and evaluating expressions (=). The abbreviation AC on a calculator stands for **A**ll **C**lear. For The Reckoner, it will clear the display. The ** operator is Python's exponentiation operator (i.e., x^y).

By clicking or tapping on the various buttons, simple or complex expressions can be displayed:



By subsequently clicking or tapping on the equal button (=), the expression can be arithmetically evaluated as follows:



The font that is used in the calculator is called **TexGyreAdventor**. It is freely available and can be obtained here: https://www.fontsquirrel.com/fonts/tex-gyre-adventor. Installing the font on the RPi is relatively simple:

(1) Download the OTF files from the Web site above; and
(2) Copy the OTF files to the proper place on the RPi's file system.

To accomplish this (particularly at the command line/terminal), you may need to refer to the first RPi activity in the curriculum: *Sampling some Raspberry Pi*.

You can either download the files directly to the RPi (e.g., into the Downloads folder) or on another system – then copy them to the RPi using a USB stick, for example. To subsequently get them to install on the RPi, simply copy them to the fonts folder from the terminal via:

```
sudo cp tex*.otf /usr/local/share/fonts
```

Then, reboot the RPi. Note that on most Linux systems (not the RPi), you can double-click the OTF files and install them that way.

In this activity, we'll take an iterative approach to creating The Reckoner. First, we'll create the GUI and ensure that it is correct before proceeding. We'll test and fix any issues before proceeding to the next

step: integrating code that will handle the button clicks/taps. As buttons are clicked/tapped, our program must accurately detect which button was clicked/tapped and, furthermore, perform some action that is appropriate for that specific button. Next, we'll work through and integrate evaluating the expression. To simplify things a bit, we'll hand that off to Python! Since it is possible that an expression has errors (e.g., mismatched parentheses, multiple consecutive operators, etc), we'll finally integrate error checking so that user errors don't unexpectedly *break* our program.

Creating the GUI

To simplify creating the GUI, we'll use Tkinter's grid manager. Recall that it allows the placement of widgets using a row/column approach. The display will be located at row 0, column 0, and span four columns. The top row of buttons ("(", ")", "AC", and "**") will be located at row 1, columns 0 through 3. The remaining buttons will be placed in increasing rows, at columns 0 through 3. Here's the calculator layout with row and column numbers:

0,0			
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3
5,0	5,1	5,2	5,3

Let's first work on a template for The Reckoner:

```
##################################################################
# Name:
# Date:
# Description:
##################################################################
from Tkinter import *

# the main GUI
class MainGUI(Frame):
    ...

##############################
# the main part of the program
##############################
# create the window
...
# set the window title
...
# generate the GUI
...
```

```
# display the GUI and wait for user interaction
...
```

Note how this simply sets up a shell that we can work with. Of course, a header will be included at the top. Since we will make use of the Tkinter library, it will have to be properly imported. Next, we'll specify the class that will represent The Reckoner's main GUI (called **MainGUI** in the code above). This class will be a Tkinter frame; therefore, it will inherit from Tkinter's **Frame** class. After the class, we'll include the main part of the program – which is pretty simple. It creates the window, sets its title, creates an instance of the **MainGUI** class (thereby generating the calculator GUI), and finally displays the GUI and waits for user interaction.

Let's continue by specifying the constructor of the **MainGUI** class and also filling in the main part of the program. New parts to the code are highlighted below:

```
class MainGUI(Frame):
    # the constructor
    def __init__(self, parent):
        Frame.__init__(self, parent, bg="white")
        self.setupGUI()

    # sets up the GUI
    def setupGUI(self):
        pass

##############################
# the main part of the program
##############################
# create the window
window = Tk()
# set the window title
window.title("The Reckoner")
# generate the GUI
p = MainGUI(window)
# display the GUI and wait for user interaction
window.mainloop()
```

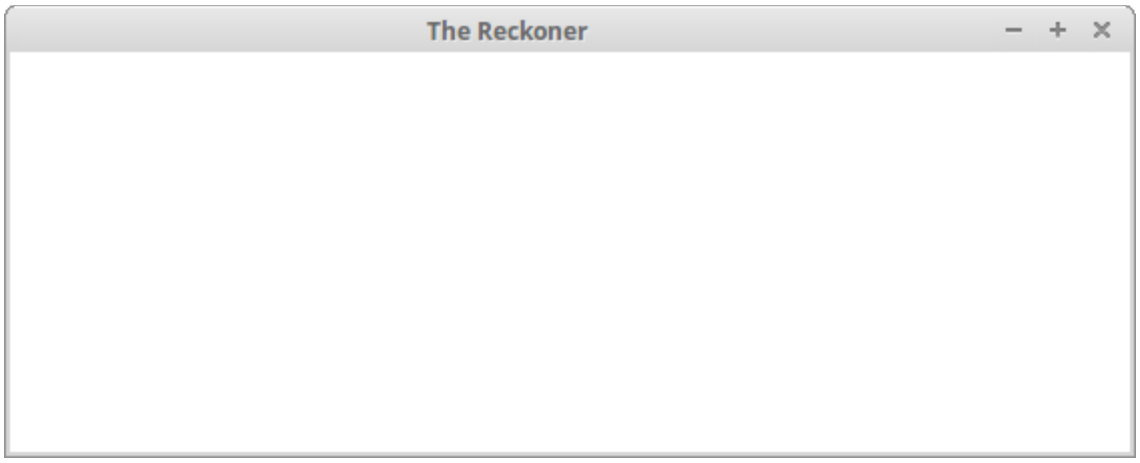
Note how the constructor merely calls the constructor of its parent **Frame** class, passing in the window and specifying the background color. Since the constructor makes use of a subroutine inside the **MainGUI** class (*setupGUI*), we can quickly stub it out by adding the **pass** statement for now. At this point, you should be able to run the program. It currently doesn't do very much; however, a small window with the title “The Reckoner” should appear:

Now, let's implement the *setupGUI* method, replacing the single **pass** statement. The display will be implemented as a Tkinter **Label**, and the buttons will each be implemented as a Tkinter **Button**. To make the interface nifty, each button will be represented with an image (GIF) that has already been designed and properly sized. Here's the first iteration of the *setupGUI* method that sets up the display:

```
    def setupGUI(self):
        # the calculator uses the TexGyreAdventor font (see
        #  https://www.fontsquirrel.com/fonts/tex-gyre-adventor)
        # on most Linux system, simply double-click the font files
        #  and install them
        # on the RPi, copy them to /usr/local/share/fonts (with
        #  sudo):
        #  sudo cp tex*.otf /usr/local/share/fonts
        # then reboot

        # the display
        # right-align text in the display; and set its background to
        #  white, its height to 2 characters, and its font to 50
        #  point TexGyreAdventor
        self.display = Label(self, text="", anchor=E, bg="white",\
         height=2, width=15, font=("TexGyreAdventor", 50))
        # put it in the top row, spanning across all four columns;
        #  and expand it on all four sides
        self.display.grid(row=0, column=0, columnspan=4,\
         sticky=E+W+N+S)

        # pack the GUI
        self.pack(fill=BOTH, expand=1)
```

Note that the backslashes are used here to delineate individual statements without ugly text wrapping in this document. In the actual code, backslashes can be omitted so that every statement is on a single line of code.

Running the updated program should now display a window with the calculator's display:



Note that Tkinter's **Label** class can be instantiated with many configuration parameters. In the case of the calculator's display, we set its text to nothing (i.e., “”); right-align the text with *anchor=E*; set its background to white; set its height to 2 characters high; set its width to 15 characters wide; and set its font to 50 point TexGyreAdventor. We then lay the display on a grid at row 0, column 0, spanning four columns, and expanding it on all sides (using the sticky configuration option). Finally, the GUI is packed such that the display fills the entire window space both horizontally and vertically.

Next, let's work on the buttons. Each button will be added to the calculator in its appropriate row and column. Since each button will be represented by an image (e.g., 0.gif, 1.gif, eql.gif, add.gif, etc), we can make use of Tkinter's **PhotoImage** class to “load” an image for each button. Button images are 115x115 pixels – which should be the perfect size for use on the RPi. The strategy will be to load the image and store it as a variable (img), create the button with the image as its property, and set the button in its proper layout position using Tkinter's grid manager. In general, here's how it's done for a button (this example uses the left parenthesis as an example and also assumes that button images are in a sub-folder called *images*):

```
img = PhotoImage(file="images/lpr.gif")
button = Button(self, bg="white", image=img)
button.image = img
button.grid(row=1, column=0, sticky=N+S+E+W)
```

The first line loads and scales the proper image, storing it to the variable *img*. The second line creates the button, setting the image as its display property. Since the button images are colored and have rounded corners, the background is set to white. The third line formally sets the button's image as the preloaded image (recall why from the lesson on GUIs). The last line places the button on the frame in its proper position along the grid, expanding it to fill its space in all directions. To make things look a little bit better, we'll additionally remove any border around the button and set the background of the button when it is active (i.e., clicked/tapped) to white.

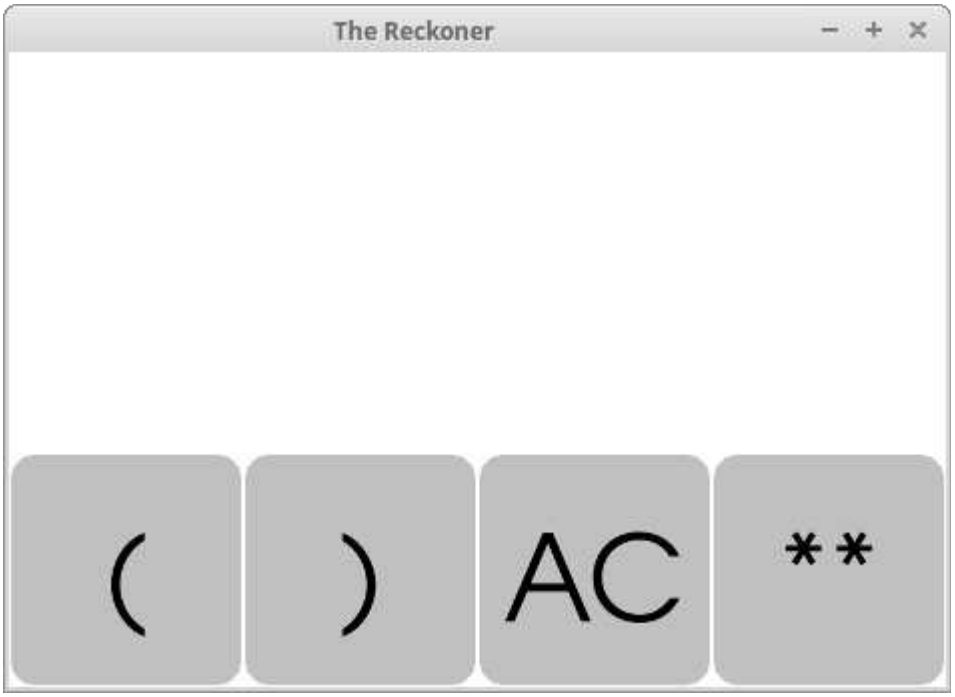
Note that the display has been slightly modified so that it no longer has the width configuration option. This is because the row of buttons will be wider than the display. By default, the display will be fitted to the new width. Here's the entire first (top) row of buttons:

```
def setupGUI(self):
    # the calculator uses the TexGyreAdventor font (see
    #  https://www.fontsquirrel.com/fonts/tex-gyre-adventor)
```

```
# on most Linux system, simply double-click the font files
#  and install them
# on the RPi, copy them to /usr/local/share/fonts (with
#  sudo):
#  sudo cp tex*.otf /usr/local/share/fonts
# then reboot

# the display
# right-align text in the display; and set its background to
#  white, its height to 2 characters, and its font to 50
#  point TexGyreAdventor
self.display = Label(self, text="", anchor=E, bg="white",\
 height=2, font=("TexGyreAdventor", 50))
# put it in the top row, spanning across all four columns;
#  and expand it on all four sides
self.display.grid(row=0, column=0, columnspan=4,\
 sticky=E+W+N+S)

# the button layout
#  (   )   AC  **
#  7   8   9   /
#  4   5   6   *
#  1   2   3   -
#  0   .   =   +

# the first row
# (
# first, fetch and store the image
# to work best on the RPi, images should be 115x115 pixels
# otherwise, may need to add .subsample(n)
img = PhotoImage(file="images/lpr.gif")
# next, create the button (white background, no border, no
#  highlighting, no color when clicked)
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
# set the button's image
button.image = img
# put the button in its proper row and column
button.grid(row=1, column=0, sticky=N+S+E+W)
# the same is done for the rest of the buttons

# )
img = PhotoImage(file="images/rpr.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=1, column=1, sticky=N+S+E+W)
# AC
```

```
img = PhotoImage(file="images/clr.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=1, column=2, sticky=N+S+E+W)
# **
img = PhotoImage(file="images/pow.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=1, column=3, sticky=N+S+E+W)

# pack the GUI
self.pack(fill=BOTH, expand=1)
```

Running the modified program should display a window with the calculator's display at the top and the first row of buttons beneath the display (in the order "(", ")", "AC", "**"):



Let's now add the remaining rows of buttons after the code for the first row (but before packing the GUI):

```
# the second row
# 7
img = PhotoImage(file="images/7.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=2, column=0, sticky=N+S+E+W)
# 8
```

```
        img = PhotoImage(file="images/8.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=2, column=1, sticky=N+S+E+W)
        # 9
        img = PhotoImage(file="images/9.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=2, column=2, sticky=N+S+E+W)
        # /
        img = PhotoImage(file="images/div.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=2, column=3, sticky=N+S+E+W)

        # the third row
        # 4
        img = PhotoImage(file="images/4.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=3, column=0, sticky=N+S+E+W)
        # 5
        img = PhotoImage(file="images/5.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=3, column=1, sticky=N+S+E+W)
        # 6
        img = PhotoImage(file="images/6.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=3, column=2, sticky=N+S+E+W)
        # *
        img = PhotoImage(file="images/mul.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white")
        button.image = img
        button.grid(row=3, column=3, sticky=N+S+E+W)

        # the fourth row
        # 1
        img = PhotoImage(file="images/1.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
```

```
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=4, column=0, sticky=N+S+E+W)
# 2
img = PhotoImage(file="images/2.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=4, column=1, sticky=N+S+E+W)
# 3
img = PhotoImage(file="images/3.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=4, column=2, sticky=N+S+E+W)
# -
img = PhotoImage(file="images/sub.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=4, column=3, sticky=N+S+E+W)

# the fifth row
# 0
img = PhotoImage(file="images/0.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=5, column=0, sticky=N+S+E+W)
# .
img = PhotoImage(file="images/dot.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=5, column=1, sticky=N+S+E+W)
# =
img = PhotoImage(file="images/eql.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=5, column=2, sticky=N+S+E+W)
# +
img = PhotoImage(file="images/add.gif")
button = Button(self, bg="white", image=img, borderwidth=0,\
 highlightthickness=0, activebackground="white")
button.image = img
button.grid(row=5, column=3, sticky=N+S+E+W)
```

```
        # pack the GUI
        self.pack(fill=BOTH, expand=1)
```

At this point, the calculator GUI is complete. The problem, however, is that it appears too large for the RPi's LCD touchscreen. There are two reasons for this: (1) the rows and columns of the grid are fixed based on the size of the button images; and (2) the rows and columns of the grid automatically expand to fit the buttons. We can set the rows and columns to resize automatically so that they all fit on the desktop by adding a few lines of code before creating the buttons:

```
        # configure the rows and columns of the Frame to adjust to
        #  the window
        # there are 6 rows (0 through 5)
        for row in range(6):
            Grid.rowconfigure(self, row, weight=1)
        # there are 4 columns (0 through 3)
        for col in range(4):
            Grid.columnconfigure(self, col, weight=1)

        # the first row
        # (
        img = PhotoImage(file="images/lpr.gif")
        ...
```

This indeed forces all of the buttons to be displayed on the desktop; however, things don't look pretty: the buttons are all cut off. This is because the calculator is taller than it is wide – and the desktop is the opposite: it is wider than it is tall. A solution is to render the calculator *sideways*! One way to do this is by rotating the RPi's display 90 degrees to the left (counter-clockwise) – or 270 degrees to the right (clockwise).

To force a rotation of the display, we need to edit the RPi's configuration from the terminal as follows:

```
sudo leafpad /boot/config.txt
```

Then, add the following line at the bottom of the file:

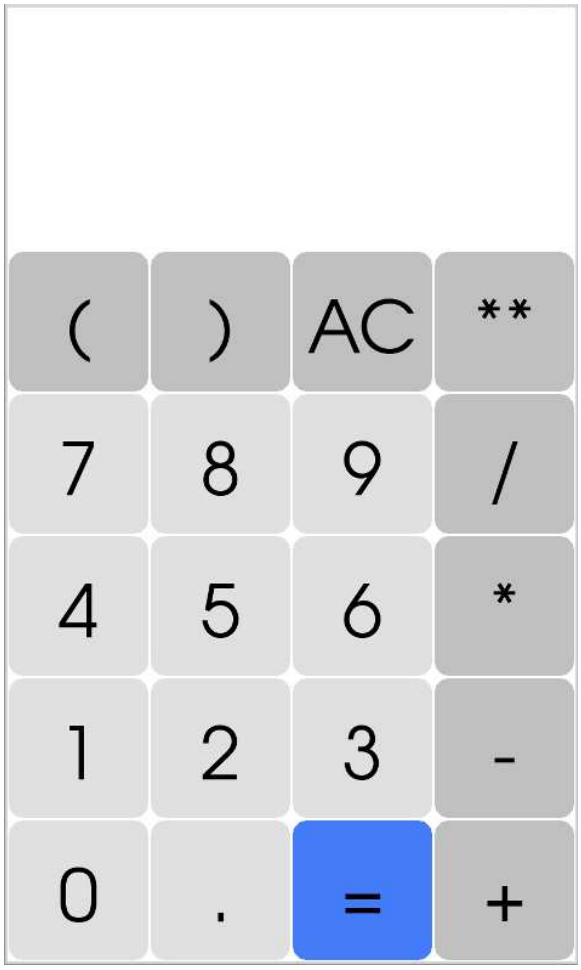
```
display_rotate=3
```

Finally, save the file (via Ctrl+S), exit the terminal, and reboot the RPi. You can set the RPi stand with its (normally) right edge on the table to right the desktop (and the calculator when the modified program is executed).

Although the calculator looks much better, it still seems a little off. This is because it was designed to be executed in fullscreen mode (i.e., without the top window bar). To force it to launch in fullscreen mode, add the following statement to the constructor of the **MainGUI** class:

```
    def __init__(self, parent):
        Frame.__init__(self, parent, bg="white")
        parent.attributes("-fullscreen", True)
        self.setupGUI()
```

The calculator should look as follows (except rendered sideways) when the program is now executed.



At this point, using the mouse to interact with the calculator works as expected. However, when using the touchscreen directly (i.e., by tapping), something seems off! The point at which a tap occurs does not correlate with the pointer on the desktop. This occurs because, although the display has been rotated, the pointer responding to taps to the touchscreen has not! That is, the touchscreen's coordinate system must also be rotated (or transformed to work along with the display's rotation). To do so, a utility called **xinput** must be installed. First, exit the calculator via Alt+F4. Then, install **xinput** via the terminal as follows (make sure that your RPi is connected to the Internet):

```
sudo apt-get update
sudo apt-get install xinput
```

Then, reboot. Subsequently, execute the following command at the terminal:

```
xinput --set-prop 'FT5406 memory based driver' 'Coordinate
 Transformation Matrix' 0 -1 1 1 0 0 0 0 1
```

Note that the command is to be entered on a single line. It is formatted for readability in this document. This transforms the touchscreen's coordinate system to one that represents a 90 degree left (counter-clockwise) rotation. To use the touchscreen in rotated mode, you will need to execute this command each time the RPi is rebooted. There is a way to automate the process; however, it is beyond the scope

of this activity.

If you wish to run the calculator on a desktop or laptop (i.e., not on the RPi), you most likely won't need to rotate the display. Furthermore, you most likely won't need to force the calculator to launch in fullscreen mode. Therefore, you can comment out the appropriate line of code in your program. If you wish to return the RPi to its normal "wide" desktop, you simply need to comment (with #) or remove the last line that was added to /boot/config.txt.

Making the buttons work

Of course, the buttons currently do nothing. Let's work on that next. In Tkinter, buttons can have a method specified that is triggered (or called) when the button is clicked. To do this, we simply need to slightly modify each button's instantiation. Here's an example with the first button in the top row (the left parenthesis):

```
            # (
            img = PhotoImage(file="images/lpr.gif")
            button = Button(self, bg="white", image=img, borderwidth=0,\
             highlightthickness=0, activebackground="white",\
             command=self.process)
            button.image = img
            button.grid(row=1, column=0, sticky=N+S+E+W)
```

Of course, this will need to be implemented for all of the buttons. In order for this to actually work, the *process* method must be implemented. We can include a simple version of this new method at the bottom of the **MainGUI** class, beneath the *setupGUI* method:

```
    def setupGUI(self):
            ...

            # pack the GUI
            self.pack(fill=BOTH, expand=1)

    # processes button presses
    def process(self):
            print "Button pressed!"
```

After implementing this for all of the buttons, running the updated program and clicking/tapping on the buttons results in "Button pressed!" being displayed to the console. On the RPi, the calculator is fullscreen mode; therefore, the console may not be visible. You can use Alt+Tab to switch through any windows on the desktop, including the console (terminal) and the calculator. Knowing when the buttons are clicked/tapped is useful; however, it is necessary to distinguish between the individual buttons (i.e., know which was clicked/tapped). A first thought may be to modify the *command* configuration option in each button's instantiation to include a parameter:

```
            # (
            img = PhotoImage(file="images/lpr.gif")
            button = Button(self, bg="white", image=img, borderwidth=0,\
             highlightthickness=0, activebackground="white",\
             command=self.process("("))
            button.image = img
```

```
        button.grid(row=1, column=0, sticky=N+S+E+W)
```

Of course, the *process* method must be updated accordingly:

```
    # processes button presses
    def process(self, button):
        print "Button {} pressed!".format(button)
```

Clearly, each button must provide its own unique parameter to the *process* method (e.g., “0” for 0, “1” for 1, “AC” for AC, “=” for =, “**” for **, “/” for /, etc). Unfortunately, running the updated program doesn't seem to work. The button clicks/taps seem to be displayed when the program is first run and not when individual buttons are actually clicked/tapped.

The problem stems from the fact that, as the buttons are instantiated, the *process* method is immediately invoked. To delay invocation of the *process* method, it must be anonymously defined using the **lambda** Python keyword. A thorough discussion of the lambda keyword is beyond the scope of this activity; however, in this case it allows the *process* method to be invoked at the time of the button click/tap. Let's modify each of the command configuration options appropriately. Here's an example of this modification to the left parenthesis (of course, the remaining buttons must be modified accordingly):

```
        # (
        img = PhotoImage(file="images/lpr.gif")
        button = Button(self, bg="white", image=img, borderwidth=0,\
         highlightthickness=0, activebackground="white",\
         command=lambda: self.process("("))
        button.image = img
        button.grid(row=1, column=0, sticky=N+S+E+W)
```

Running the modified program should display text appropriate to a clicked/tapped button (as it is clicked/tapped).

The next step is to update the *process* method so that it actually behaves properly depending on the button that is clicked/tapped. We can implement simple logic to accomplish this. Almost all of the buttons, when clicked/tapped, should just be added to the display. The two exceptions include the AC and equals (=) buttons. AC should clear the display, and = should evaluate the expression and update the display with the result. Let's start with the AC button since it's perhaps the simplest case. Change the *process* method as follows:

```
    # processes button presses
    def process(self, button):
        # AC clears the display
        if (button == "AC"):
            # clear the display
            self.display["text"] = ""
```

The display is cleared by changing its *text* attribute. In Python, associative arrays (i.e., arrays with named indexes as opposed to numeric indexes) are valid. Although you probably haven't seen associative arrays yet, you will later in the curriculum (Python calls them dictionaries).

The next case is to handle the various buttons that should just be appended to the display. Modify the

process method as follows:

```
    # processes button presses
    def process(self, button):
        # AC clears the display
        if (button == "AC"):
            # clear the display
            self.display["text"] = ""
        # otherwise, just tack on the appropriate operand/operator
        else:
            self.display["text"] += button
```

Notice how the last line appends the button parameter to the text already on the display. Running the modified program now allows the calculator to behave as expected. The AC button clears the display, while the other buttons (including, for now, the equals button) are appended to the display in the order that they are clicked/tapped. Of course, this is not the intended behavior of the equals button!

Evaluating expressions

Finally, we must discuss the equals (=) button. How, exactly, should this work? Admittedly, we could manually calculate the expression and return a result. However, Python provides the *eval* function that takes a string, evaluates it as an arithmetic expression, and returns a numeric result. Here are several examples:

```
    eval("1+1")              # the returned result is 2
    eval("5+(4-2)**5+6*2")   # the returned result is 49
```

Evaluating the expression on the display is now pretty simple! We just need to send the expression to Python's *eval* function to get a numeric result. Then, we can change the display, replacing it with a string version of the returned result. It looks something like this:

```
        expr = self.display["text"]
        result = eval(expr)
        self.display["text"] = str(result)
```

The first line stores the expression from the display to the variable *expr*, the second line evaluates the expression with Python's *eval* function, and the third line stores the result (as a string) back to the display. Of course, the three lines could be combined into a single statement; however, it is probably slightly less readable:

```
        self.display["text"] = str(eval(self.display["text"]))
```

Let's implement the evaluation portion by modifying the *process* method as follows:

```
    # processes button presses
    def process(self, button):
        # AC clears the display
        if (button == "AC"):
            # clear the display
            self.display["text"] = ""
        # = starts an evaluation of whatever is on the display
        elif (button == "="):
            # get the expression in the display
```

```
            expr = self.display["text"]
            # evaluate the expression
            result = eval(expr)
            # store the result to the display
            self.display["text"] = str(result)
        # otherwise, just tack on the appropriate operand/operator
        else:
            self.display["text"] += button
```

At this point, a valid arithmetic expression in the display will be properly evaluated and updated with the result. But what if the expression in the display is **not** valid? For example, what is the result of the expression 4 + 5 * (note the missing operand after the multiplication operator)? In fact, an error is outputted to the console (note that errors on different systems may differ slightly):

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "/usr/lib/python2.7/lib-tk/Tkinter.py", line 1489, ...
    return self.func(*args)
  File "TheReckoner-TEMPLATE9.py", line 132, in <lambda>
    button = Button(self, bg="white", image=img, command=...
  File "TheReckoner-TEMPLATE9.py", line 154, in process
    result = eval(expr)
  File "<string>", line 1
    4+5*
       ^
SyntaxError: unexpected EOF while parsing
```

Evidently, there was an unexpected EOF (end of file) while parsing the expression. Of course, this makes sense: the end of the expression was reached before a valid expression was provided. That is, the *eval* function expected more to the expression. To make the calculator more robust, we can detect such errors and provide an appropriate response to the user. This can be accomplished by using a **try-except** block (you should have seen this before!). The purpose of a **try-except** block is to encapsulate instructions that could cause an exception (something that alters the normal flow of a program) in a **try** block. If an exception occurs, it can be handled in the **except** block. More details about **try-except** will be covered later in the curriculum.

Let's handle any invalid expression evaluation by setting the calculator's display to the string ERROR so that the user is aware that an error occurred. This can be accomplished by modifying the *process* method as follows:

```
            # get the expression in the display
            expr = self.display["text"]
            # the evaluation may return an error!
            try:
                # evaluate the expression
                result = eval(expr)
                # store the result to the display
                self.display["text"] = str(result)
            # handle if an error occurs during evaluation
```

```
            except:
                # note the error in the display
                self.display["text"] = "ERROR"
```

Note that the evaluation of the expression (into the variable *result*) and the subsequent modification of the display with the result is done in the **try** block. If this results in an error, the **except** block is executed, setting the contents of the display to the string ERROR.

Congratulations, you've made a simple calculator! Indeed, it's basic; however, there are many improvements that could be made:

- The length of the display supports approximately 12 characters; however, more characters could be added to the display by the user. Doing so results in a cropped expression, which is probably not desired. The display could be limited to 12 characters, ignoring any further input. Moreover, the result of an expression could be larger than 12 characters. Perhaps such results could be truncated.
- There is no easy way to erase the last character entered of an expression in the display. Adding a back button to the calculator could allow this in case the user makes an error.
- Other useful operators could be added. For example: modulus (%), square root, logarithm (log), natural logarithm (ln), factorial (!), sine, cosine, tangent, and constants such as pi and e. Take look at the Google calculator (search for "calculator" on Google). Of course, Python's *eval* function must support these (you'll have to try them out!).
- Most calculators clear any previous expression result if the user enters a new expression. For example, suppose that the display has the result of an expression. If the user decides to enter a different expression, the display must first be cleared. Currently, the user must do this manually (by pressing the AC button). Perhaps the display could be automatically cleared of an expression result if the user begins to enter a new expression.
- Similarly, perhaps the display could be cleared of an error if the user begins to enter a new expression.

For completeness, here's the entire code (reduced in font size for readability):

```
from Tkinter import *

# the main GUI
class MainGUI(Frame):
    # the constructor
    def __init__(self, parent):
        Frame.__init__(self, parent, bg="white")
        parent.attributes("-fullscreen", True)
        self.setupGUI()

    # sets up the GUI
    def setupGUI(self):
        # the calculator uses the TexGyreAdventor font (see
        # https://www.fontsquirrel.com/fonts/tex-gyre-adventor)
        # on most Linux system, simply double-click the font
        #  files and install them
        # on the RPi, copy them to /usr/local/share/fonts (with
```

```
#  sudo):
#  sudo cp tex*.otf /usr/local/share/fonts
# then reboot

# the display
# right-align text in the display; and set its
#  background to white, its height to 2 characters, and
#  its font to 50 point TexGyreAdventor
self.display = Label(self, text="", anchor=E,\
 bg="white", height=2, font=("TexGyreAdventor", 50))
# put it in the top row, spanning across all four
#  columns; and expand it on all four sides
self.display.grid(row=0, column=0, columnspan=4,\
 sticky=E+W+N+S)

# the button layout
#  (   )   AC  **
#  7   8   9   /
#  4   5   6   *
#  1   2   3   -
#  0   .   =   +

# configure the rows and columns of the Frame to adjust
#  to the window
# there are 6 rows (0 through 5)
for row in range(6):
     Grid.rowconfigure(self, row, weight=1)
# there are 4 columns (0 through 3)
for col in range(4):
     Grid.columnconfigure(self, col, weight=1)

# the first row
# (
# first, fetch and store the image
# to work best on the RPi, images should be 115x115
#  pixels
# otherwise, may need to add .subsample(n)
img = PhotoImage(file="images/lpr.gif")
# next, create the button (white background, no border,
#  no highlighting, no color when clicked)
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("("))
# set the button's image
button.image = img
# put the button in its proper row and column
button.grid(row=1, column=0, sticky=N+S+E+W)
```

```
# the same is done for the rest of the buttons
# )
img = PhotoImage(file="images/rpr.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process(")"))
button.image = img
button.grid(row=1, column=1, sticky=N+S+E+W)
# AC
img = PhotoImage(file="images/clr.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("AC"))
button.image = img
button.grid(row=1, column=2, sticky=N+S+E+W)
# **
img = PhotoImage(file="images/pow.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("**"))
button.image = img
button.grid(row=1, column=3, sticky=N+S+E+W)

# the second row
# 7
img = PhotoImage(file="images/7.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("7"))
button.image = img
button.grid(row=2, column=0, sticky=N+S+E+W)
# 8
img = PhotoImage(file="images/8.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("8"))
button.image = img
button.grid(row=2, column=1, sticky=N+S+E+W)
# 9
img = PhotoImage(file="images/9.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
```

```
 self.process("9"))
button.image = img
button.grid(row=2, column=2, sticky=N+S+E+W)
# /
img = PhotoImage(file="images/div.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("/"))
button.image = img
button.grid(row=2, column=3, sticky=N+S+E+W)

# the third row
# 4
img = PhotoImage(file="images/4.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("4"))
button.image = img
button.grid(row=3, column=0, sticky=N+S+E+W)
# 5
img = PhotoImage(file="images/5.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("5"))
button.image = img
button.grid(row=3, column=1, sticky=N+S+E+W)
# 6
img = PhotoImage(file="images/6.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("6"))
button.image = img
button.grid(row=3, column=2, sticky=N+S+E+W)
# *
img = PhotoImage(file="images/mul.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("*"))
button.image = img
button.grid(row=3, column=3, sticky=N+S+E+W)

# the fourth row
# 1
```

```
img = PhotoImage(file="images/1.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("1"))
button.image = img
button.grid(row=4, column=0, sticky=N+S+E+W)
# 2
img = PhotoImage(file="images/2.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("2"))
button.image = img
button.grid(row=4, column=1, sticky=N+S+E+W)
# 3
img = PhotoImage(file="images/3.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("3"))
button.image = img
button.grid(row=4, column=2, sticky=N+S+E+W)
# -
img = PhotoImage(file="images/sub.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("-"))
button.image = img
button.grid(row=4, column=3, sticky=N+S+E+W)

# the fifth row
# 0
img = PhotoImage(file="images/0.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("0"))
button.image = img
button.grid(row=5, column=0, sticky=N+S+E+W)
# .
img = PhotoImage(file="images/dot.gif")
button = Button(self, bg="white", image=img,\
 borderwidth=0, highlightthickness=0,\
 activebackground="white", command=lambda:\
 self.process("."))
button.image = img
```

```
            button.grid(row=5, column=1, sticky=N+S+E+W)
            # =
            img = PhotoImage(file="images/eql.gif")
            button = Button(self, bg="white", image=img,\
             borderwidth=0, highlightthickness=0,\
             activebackground="white", command=lambda:\
             self.process("="))
            button.image = img
            button.grid(row=5, column=2, sticky=N+S+E+W)
            # +
            img = PhotoImage(file="images/add.gif")
            button = Button(self, bg="white", image=img,\
             borderwidth=0, highlightthickness=0,\
             activebackground="white", command=lambda:\
             self.process("+"))
            button.image = img
            button.grid(row=5, column=3, sticky=N+S+E+W)

            # pack the GUI
            self.pack(fill=BOTH, expand=1)

      # processes button presses
      def process(self, button):
            # AC clears the display
            if (button == "AC"):
                  # clear the display
                  self.display["text"] = ""
            # = starts an evaluation of whatever is on the display
            elif (button == "="):
                  # get the expression in the display
                  expr = self.display["text"]
                  # the evaluation may return an error!
                  try:
                        # evaluate the expression
                        result = eval(expr)
                        # store the result to the display
                        self.display["text"] = str(result)
                  # handle if an error occurs during evaluation
                  except:
                        # note the error in the display
                        self.display["text"] = "ERROR"
            # otherwise, just tack on the appropriate
            #  operand/operator
            else:
                  self.display["text"] += button

##############################
# the main part of the program
```

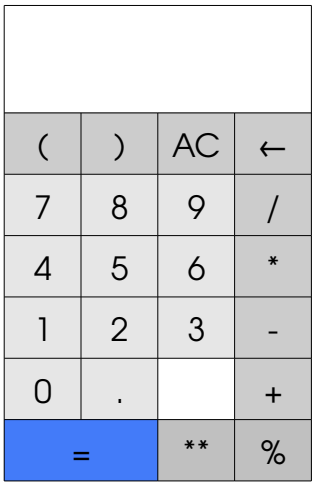
```
##############################
# create the window
window = Tk()
# set the window title
window.title("The Reckoner")
# generate the GUI
p = MainGUI(window)
# display the GUI and wait for user interaction
window.mainloop()
```

Homework: The Reckoner

For the homework portion of this activity, you may have the option to work in **groups** (pending prof approval). It is suggested that groups contain at least one confident Python coder.

Your task is to implement the following improvements to the calculator:

(1) Add a modulus (%) button that calculates the remainder returned by a division. For example: 23 % 13 = 10.
(2) Add a back button that removes the last (i.e., right-most) character in the display.
(3) Modify the layout as follows to accommodate the new buttons:



See below for what that should look like on the RPi.

(4) Limit the display to 14 characters. Do not allow the user to enter more than 14 characters.
(5) Any result that is greater than 14 characters should be truncated to the first 11 characters followed by three successive dots. For example: 2 ** 47 = 14073748835...
(6) If an expression result (or an error) has just been put on the display, clear the screen before displaying the next character inputted by the user.

A few notes:

- The calculator has now increased by one row. This has the unfortunate effect of no longer properly rendering the calculator on the LCD touchscreen. To fix this, we can reduce the font size of text in the display from 50 point to 45 point. Furthermore, we can reduce the height of the display from two characters to one character.

- The equals (=) button now spans two columns. You will have to download the new image for this button.
- There is now a blank "button" on the calculator. Feel free to just leave this "button" blank.

Do not make any additional improvements to your submission for this assignment. However, feel free to do so on your own if you wish (although you may need to recreate the buttons if you choose to add more rows or columns to the calculator).

You are to submit your Python source code only (as a .py file) through the upload facility on the web site.

Raspberry Pi Activity: Simon

In this activity, you will implement a game that is similar to the popular (well, years ago anyways) game called Simon. Simon is an electronic memory skill game. Here's an image of the game as manufactured by Milton Bradley:

The game *board* is circular and has four large buttons that light up, each of a different color. Each color button has a musical note or tone associated with it. The game begins by randomly picking one (or sometimes two or even three) random colors. These randomly chosen colors are called a sequence. The game then plays the sequence by lighting up the appropriate colored buttons and playing the corresponding notes. The player then tries to replicate the sequence exactly. Any mistake, and the game ends. Each time the player successfully plays a sequence and matches the randomly selected colors, the sequence grows by an extra color.

For this activity, you will need the following items:

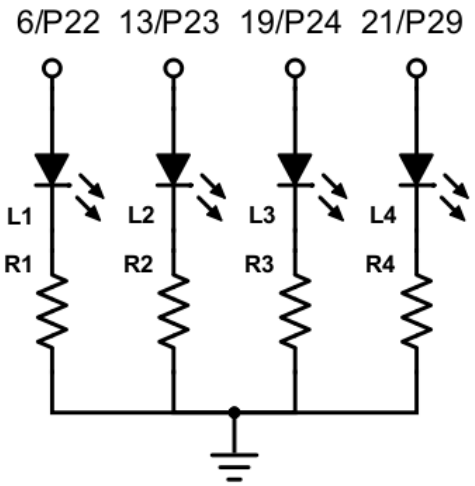
- Raspberry Pi B v3 with power adapter;
- LCD touchscreen;
- Keyboard and mouse;
- USB-powered speakers;
- Breadboard;
- GPIO interface board with ribbon cable; and
- LEDs, resistors, switches, and jumper wires provided in your kit.

Regarding the electronic components, you will need the following:

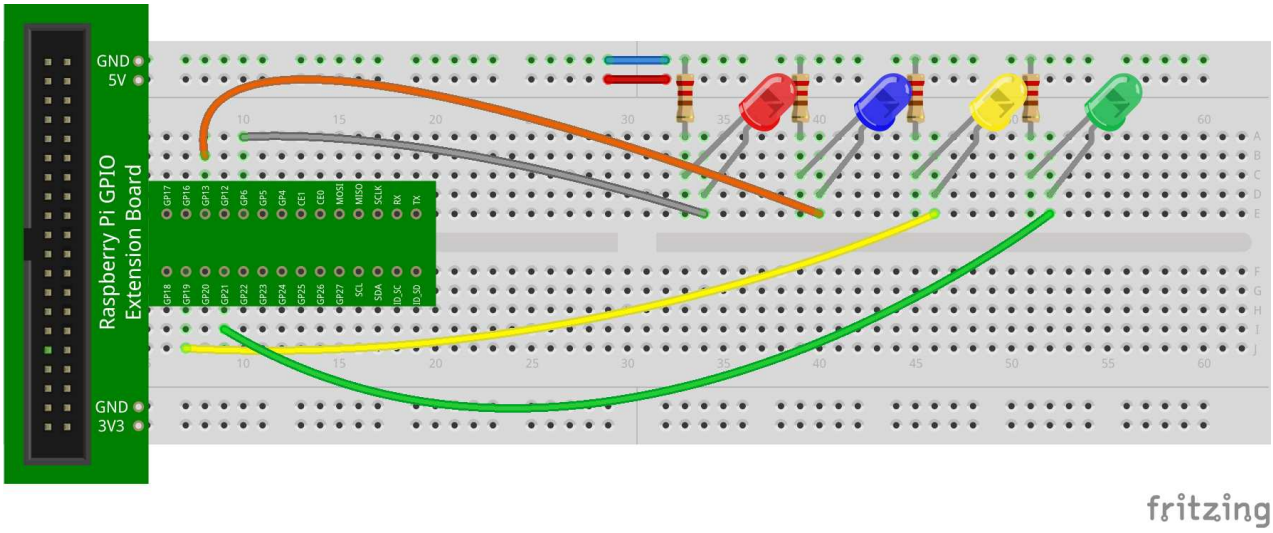
- 1x red LED;
- 1x blue LED;
- 1x yellow LED;
- 1x green LED;

- 4x push-button switches;
- 4x 220Ω resistors; and
- 16x jumper wires.

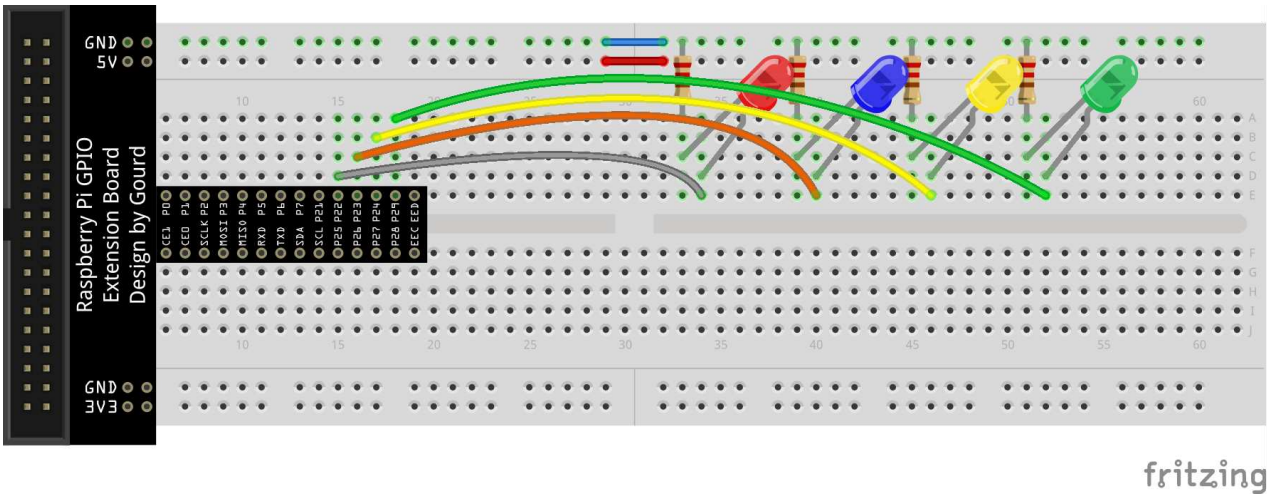
The circuit
To begin, implement the following circuit:



Although you can wire the LEDs to different GPIO pins on the RPi, it will be easier if you follow the circuit diagram shown above because it will match the source code provided. Also, try to keep enough space for four push-button switches in between the GPIO-to-breadboard interface and the LEDs. This will make it easier to implement the other parts of this activity. Here's one way to layout the circuit:



If you have the black GPIO interface, layout the circuit as follows instead:



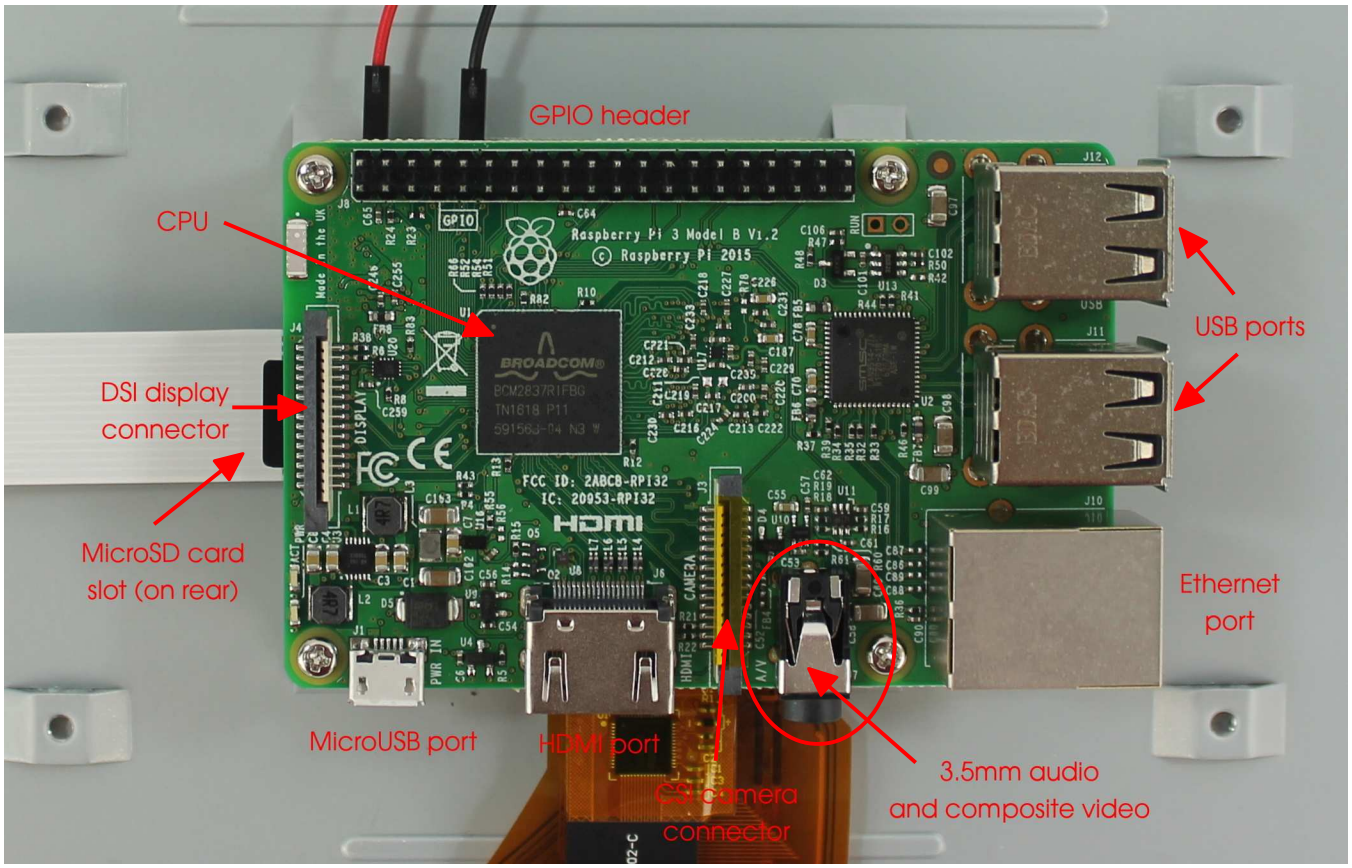
For this part of the activity, you will simply turn the LEDs on, one at a time. As each LED is turned on, a corresponding note will play. You will use the Pygame library to play the notes. Pygame is a set of Python libraries that are useful for making games. For this activity, you will make use of its multimedia support (specifically, the ability to play sound files).

To play the notes, you will first need to obtain four sound files located on the class web site:

- `one.wav`
- `two.wav`
- `three.wav`
- `four.wav`

It is recommended that you create a folder for this activity and place the sound files there.

To hear the notes being played, you will need to use the USB-powered speakers. Connect the USB cable from the speakers to an open USB port on the RPi, and connect the audio cable from the speakers to the audio jack as shown below:



Turning the LEDs on and making noise!

Either using IDLE or a text editor, type the following Python code and save it to a file in the same folder as the sound files that you downloaded and saved earlier:

```
 1   import RPi.GPIO as GPIO
 2   from time import sleep
 3   import pygame

 4   # initialize the pygame library
 5   pygame.init()

 6   # set the GPIO pin numbers
 7   # the LEDs (from L to R)
 8   leds = [6, 13, 19, 21 ]
 9   # the sounds that map to each LED (from L to R)
10   sounds = [ pygame.mixer.Sound("one.wav"),
     pygame.mixer.Sound("two.wav"),
     pygame.mixer.Sound("three.wav"),
     pygame.mixer.Sound("four.wav") ]

11   # use the Broadcom pin mode
12   GPIO.setmode(GPIO.BCM)

13   # setup the output pins
```

```
14   GPIO.setup(leds, GPIO.OUT)

15   print "Watch the LEDs light with sound!"
16   for i in range(len(leds)):
17        # light the current LED
18        GPIO.output(leds[i], True)
19        # play its corresponding sound
20        sounds[i].play()
21        # wait a bit, then turn the LED off
22        sleep(1)
23        GPIO.output(leds[i], False)
24        sleep(0.5)
25   print "Sionara!"
26   GPIO.cleanup()
```

Let's explain the program. In lines 1 through 3, required libraries are imported. For this activity, we need GPIO functionality (since we're turning on LEDs), the sleep function (to implement delays), and the Pygame library (to play the sound files).

In order to to use the modules in the Pygame library, it must first be initialized. This is done in line 5.

The next step is to setup the GPIO output pins that are wired to the LEDs in a list (in line 8). For this activity, the LEDs are wired to pins GP6= P22 (red), GP13= P23 (blue), GP19= P24 (yellow), and GP21= P29 (green). The sound files are also defined in a list and preloaded for later use (in line 10). In lines 11 through 14, the GPIO pin mode is specified, and the GPIO pins wired to the LEDs are setup as output pins.

The remainder of the source code (lines 15 through 25) turns each LED on, one at a time, plays each LED's corresponding sound file, waits a few moments, and turns the LED off. If you need a refresher on GPIO in Python, it is suggested that you go back to Raspberry Pi Activity 2: My Binary Addiction...Reloaded. Line 26 cleans up the GPIO pins ands resets them to their defaults.

Get this part working before going on to the next part of the activity. Make sure that you see the LEDs turning on and off and hear the notes playing as each LED is briefly turned on. If the speakers aren't working, you can try the following:

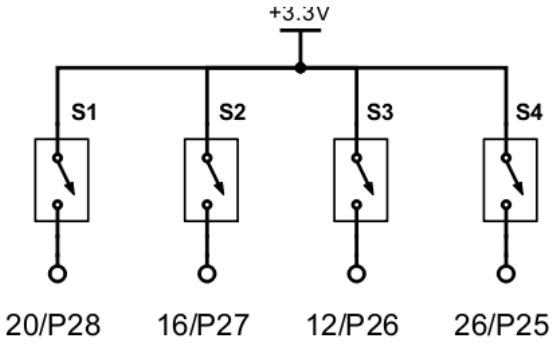
(1) Make sure that both the speaker's USB and audio cables are plugged in.
(2) Make sure that the sound files are in the same folder as your `.py` source file.
(3) Make sure that the sound files are spelled correctly (as they are named in the folder) in your source code. Remember that **filenames are cl se sensitive**!
(4) Make sure that the audio configuration on the RPi is set to output to the analog 3.5mm (headphone) jack. To do this, right-click on the speaker icon at the upper right of the desktop and select **Analog**.
(5) Make sure that the volume *wheel* on the back of one of the speakers is turned to the left (not quite all the way) and that the volume on the RPi is close (but not all the way) to its maximum (click on the speaker icon to set the volume on the RPi).
(6) If you still have problems, open up a terminal (by clicking on the monitor icon at the upper left of the desktop) and type `amixer set PCM -- 100%`.

(7) If you can hear the notes but they seem broken (e.g., with pops and clicks), you may need to turn the volume down on the speakers. Do this by sliding the volume *wheel* on the back of one of the speakers to the right a little until the notes are *clean*. Another option is to reduce the volume on the RPi by modifying the percentage value in the following terminal command: `amixer set PCM – 100%`. Try changing it to 85%, 75%, and so on, until the notes are *clean*.

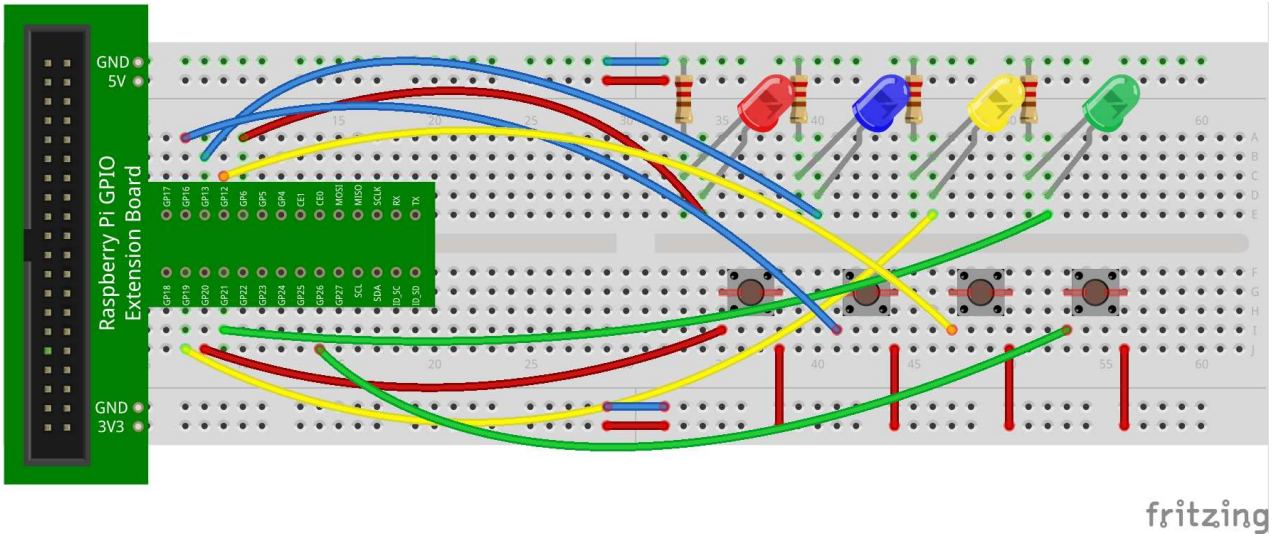
Adding switches

Extend your circuit to include four push-button switches. For this part of the activity, you will modify the previous circuit so that four push-button switches control the four LEDs. Pushing on the switches will turn on the appropriate LEDs and play the corresponding notes.

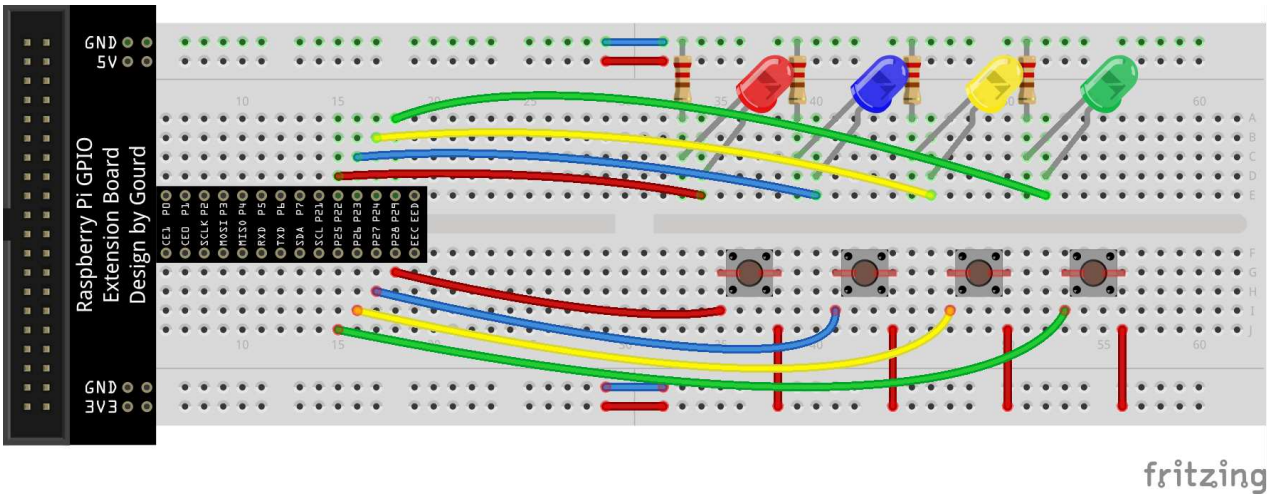
Add four switches to your circuit as show below:



Here's one way to layout this circuit:



If you have the black GPIO interface, layout the circuit as follows instead:



Make sure that the switches are wired to +3.3V on one side and to an appropriate GPIO pin on the other side (GP26=P25, GP12=P26, GP16=P27, GP20= P28 in the figures above). The input pins will be pulled down (i.e., 0V) by default, and pushing on the switches will drive the input pins high. The goal will be to detect when this occurs so that the appropriate LED can be turned on.

You will now create a new Python program. Make sure that it is also saved in the same folder as the sound files that were downloaded earlier. In IDLE or a text editor, type in the following new program:

```
 1   import RPi.GPIO as GPIO
 2   from time import sleep
 3   import pygame

 4   # initialize the pygame library
 5   pygame.init()

 6   # set the GPIO pin numbers
 7   # the switches (from L to R)
 8   switches = [ 20, 16, 12, 26 ]
 9   # the LEDs (from L to R)
10   leds = [6, 13, 19, 21 ]
11   # the sounds that map to each LED (from L to R)
12   sounds = [ pygame.mixer.Sound("one.wav"),
     pygame.mixer.Sound("two.wav"),
     pygame.mixer.Sound("three.wav"),
     pygame.mixer.Sound("four.wav") ]

13   # use the Broadcom pin mode
14   GPIO.setmode(GPIO.BCM)

15   # setup the input and output pins
16   GPIO.setup(switches, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
17   GPIO.setup(leds, GPIO.OUT)

18   print "Press the switches or Ctrl+C to exit..."
```

```
19   # we'll discuss this later, but this allows us to detect
20   # when Ctrl+C is pressed so that we can reset the GPIO pins
21   try:
22        # keep going until the user presses Ctrl+C
23        while (True):
24             # initially note that no switch is pressed
25             # this will help with switch debouncing
26             pressed = False
27             # so long as no switch is currently pressed...
28             while (not pressed):
29                  # ...we can check the status of each switch
30                  for i in range(len(switches)):
31                       # if one switch is pressed
32                       while (GPIO.input(switches[i]) == True):
33                            # note its index
34                            val = i
35                            # note that a switch has now been
                                pressed
36                            pressed = True

37             # light the matching LED
38             GPIO.output(leds[val], True)
39             # play its corresponding sound
40             sounds[val].play()
41             # wait and turn the LED off again
42             sleep(1)
43             GPIO.output(leds[val], False)
44             sleep(0.25)
45   # detect Ctrl+C
46   except KeyboardInterrupt:
47        # reset the GPIO pins
48        GPIO.cleanup()
49        print "\nSionara!"
```

You should notice that some of this new program is similar to the previous one. The first difference is that a new list is defined that stores the GPIO pins that are wired to the push-button switches (in line 8). Make sure that the specified GPIO pins match the connections on your breadboard.

Since we now have input pins, they need to be defined as such. This is done in line 16. Note that the input pins are pulled down by default.

Note the **try-except** construct (lines 21 and 46). Although you should have seen this before, it hasn't yet been thoroughly explained. Essentially, a try-except construct encapsulates any part of a Python program that could potentially experience abnormal program behavior. In this case, the goal is to detect when a user presses Ctrl+C. If this key combination is detected, we wish to reset the GPIO pins (thereby resetting them to their defaults), and exit the program (lines 48 through 49).

The goal of the program is to continually wait for a switch to be pressed. When one is pressed, the program tries to detect which one it is, and light the appropriate LED. After a brief moment, the program should wait for another switch to be pressed. The only way to end the program is to press Ctrl+C. To check for switch presses indefinitely, the program uses a while loop. Note the condition of this while loop in line 23. Since the condition is always true, the while loop executes forever! But this is OK, since we are allowing Ctrl+C to abort and exit the program.

When reading switch presses, we must worry about an issue called debouncing. **Switch debouncing** prevents a single press of a push-button switch from appearing like multiple presses. This is something that we generally have to live with when using switches in digital circuits. The tactic in this program is to utilize a Boolean variable called `pressed` that detects when any one of the switches is pressed. Initially, it is set to false (i.e., no switch is pressed) in line 26. The while loop beginning at line 28 is then entered. When any one of the switches is pressed, the variable is toggled to true (line 36). This breaks control out of the while loop, allowing the appropriate LED to be turned on and the corresponding note to be played (lines 37 through 44). Since the variable is toggled to true, then no other switch press can be detected until it is reset to false.

Detecting which switch is pressed (if any), is done in the for loop beginning at line 30. The program checks the state of each switch, one by one (line 32). While any switch is pressed (i.e., its wired input pin is high), its index is noted (line 34). A switch's index corresponds to the index of the LED in the list of LEDs that it controls (and the sound file that should be played). Once a switch press has been detected, the variable `pressed` is set to true in line 36 (which breaks out of the while loop beginning at line 28).

Again, once a switch has been pressed, the appropriate LED is turned on, and the corresponding sound file is played (lines 37 through 40). After a brief moment, the LED is turned back off (lines 41 through 44), and the outer while loop beginning at line 23 begins again.

Get this part working before going on to the next part of the activity. Make sure that you see the LEDs turning on and off as you press the switches. Make sure that you hear the notes playing as each LED is briefly turned on.

Simon

The parts of the activity that you have already implemented provide almost all that is needed to lay the base for the game. The only thing left to add is a way to generate a random sequence of colors, allow the player to push buttons that correspond the to sequence, check if the player's sequence matches the one in the game, and either grow the sequence or end the game!

First, here's the code:

```
1  import RPi.GPIO as GPIO
2  from time import sleep
3  from random import randint
4  import pygame

5  # set to True to enable debugging output
6  DEBUG = False
```

```
 7  # initialize the pygame library
 8  pygame.init()

 9  # set the GPIO pin numbers
10  # the switches (from L to R)
11  switches = [ 20, 16, 12, 26 ]
12  # the LEDs (from L to R)
13  leds = [6, 13, 19, 21 ]
14  # the sounds that map to each LED (from L to R)
15  sounds = [ pygame.mixer.Sound("one.wav"),
    pygame.mixer.Sound("two.wav"),
    pygame.mixer.Sound("three.wav"),
    pygame.mixer.Sound("four.wav") ]

16  # use the Broadcom pin mode
17  GPIO.setmode(GPIO.BCM)

18  # setup the input and output pins
19  GPIO.setup(switches, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
20  GPIO.setup(leds, GPIO.OUT)

21  # this function turns the LEDs on
22  def all_on():
23       for i in leds:
24            GPIO.output(leds, True)

25  # this function turns the LEDs off
26  def all_off():
27       for i in leds:
28            GPIO.output(leds, False)

29  # this functions flashes the LEDs a few times when the
      player loses the game
30  def lose():
31       for i in range(0, 4):
32            all_on()
33            sleep(0.5)
34            all_off()
35            sleep(0.5)

36  # the main part of the program
37  # initialize the Simon sequence
38  # each item in the sequence represents an LED (or switch),
      indexed at 0 through 3
39  seq = []
40  # randomly add the first two items to the sequence
41  seq.append(randint(0, 3))
```

```
42  seq.append(randint(0, 3))

43  print "Welcome to Simon!"
44  print "Try to play the sequence back by pressing the
    switches."
45  print "Press Ctrl+C to exit..."

46  # we'll discuss this later, but this allows us to detect
47  # when Ctrl+C is pressed so that we can reset the GPIO pins
48  try:
49       # keep going until the user presses Ctrl+C
50       while (True):
51            # randomly add one more item to the sequence
52            seq.append(randint(0, 3))
53            if (DEBUG):
54                 # display the sequence to the console
55                 if (len(seq) > 3):
56                      print
57                 print "seq={}".format(seq)

58            # display the sequence using the LEDs
59            for s in seq:
60                 # turn the appropriate LED on
61                 GPIO.output(leds[s], True)
62                 # play its corresponding sound
63                 sounds[s].play()
64                 # wait and turn the LED off again
65                 sleep(1)
66                 GPIO.output(leds[s], False)
67                 sleep(0.5)

68            # wait for player input (via the switches)
69            # initialize the count of switches pressed to 0
70            switch_count = 0
71            # keep accepting player input until the number of
                items in the sequence is reached
72            while (switch_count < len(seq)):
73                 # initially note that no switch is pressed
74                 # this will help with switch debouncing
75                 pressed = False
76                 # so long as no switch is currently
                     pressed...
77                 while (not pressed):
78                      # ...we can check the status of each
                          switch
79                      for i in range(len(switches)):
80                           # if one switch is pressed
81                           while (GPIO.input(switches[i]) ==
```

```
                                True):
 82                                  # note its index
 83                                  val = i
 84                                  # note that a switch has now
                                       been pressed
 85                                # so that we don't detect any more
                                     switch presses
 86                                  pressed = True

 87              if (DEBUG):
 88                   # display the index of the switch
                        pressed
 89                   print val,

 90              # light the matching LED
 91              GPIO.output(leds[val], True)
 92              # play its corresponding sound
 93              sounds[val].play()
 94              # wait and turn the LED off again
 95              sleep(1)
 96              GPIO.output(leds[val], False)
 97              sleep(0.25)

 98              # check to see if this LED is correct in the
                   sequence
 99              if (val != seq[switch_count]):
100                   # player is incorrect; invoke the lose
                        function
101                   lose()
102                   # reset the GPIO pins
103                   GPIO.cleanup()
104                   # exit the game
105                   exit(0)

106              # if the player has this item in the sequence
                   correct, increment the count
107              switch_count += 1
108  # detect Ctrl+C
109  except KeyboardInterrupt:
110       # reset the GPIO pins
111       GPIO.cleanup()
```

To support randomly generating the sequence, the random library is imported in line 3. Also, it is often useful to display debugging information while developing applications. In the case of Simon, it is useful to show the randomly generated sequence, and the player's submitted sequence. This can help during testing. The debug variable is set in line 6. When it is set to false, no debugging information is shown.

Lines 7 through 20 are the same as the last part of this activity. Lines 21 through 35 define three new functions. The first simply turns all of the LEDs on. The second turns all of the LEDs off. The third is invoked when the player loses the game. This function blinks all of the LEDs a few times before the game ends.

Lines 39 through 42 setup the empty sequence and add two random colors to the sequence. After some short introduction text, the game officially begins with the while loop beginning at line 50.

Each time the while loop iterates, a new random color is added to the sequence (line 52). If debugging is enabled, the sequence is then displayed to the console (lines 55 through 57). The game then displays the sequence, turning on the appropriate LEDs and playing the corresponding notes, one by one (lines 59 through 67).

Since we now want to detect the pressing of multiple switches sequentially (i.e., we need to detect the player's submission for the entire sequence), a counter is initialized (line 70) and is increased each time the player presses a switch (later in line 107). The program accepts as many switch presses as there are colors in the sequence (line 72). The detection of which switch is pressed is the same as in the previous part of the activity (lines 75 through 86). If debugging is enabled, the index of the pressed switch is displayed to the console (line 89).

Lines 90 through 97 turn the LED on that corresponds to the player's switch press, plays the corresponding note, waits a brief moment, then turns the LED off. Line 99 then checks to make sure that the player's switch press is indeed the right one in the sequence. If not, the player loses and the game ends (lines 101 through 105).

Homework: Simon

For the homework portion of this activity, you may have the option to work in **groups** (pending prof approval). It is suggested that groups contain at least one confident Python coder.

The **first part** of this activity is to implement the Simon program in this activity. Please work to understand the algorithm and source code instead of merely typing it in (or, worse, using a copy/paste process). Once your Simon game is working properly, you can move on the the second part of this activity.

For the **second part** of this activity, you must implement several improvements:

(1) **A scoring mechanism.** When the player makes a mistake and the game ends, output a message similar to, "You made it to a sequence of 9!" Remember that failing at the start should output something like, "You made it to a sequence of 0!" or, "You didn't even make it to a sequence!"

(2) **Over time, increase the speed of the playing sequence.** As the player is more and more successful, increase the speed of the sequence as it is played back to the player. The time spent playing each note in a sequence is currently 1s. Furthermore, the delay in between playing the notes is currently 0.5s. Modify this as follows:

1. Once the sequence gets to **five notes**, the time spent playing each each note should be decreased to **0.9s**; the delay in between playing the notes should be decreased to **0.4s**.
2. Once the sequence gets to **seven notes**, the time spent playing each each note should be decreased to **0.8s**; the delay in between playing the notes should be decreased to **0.3s**.

3. Once the sequence gets to **ten notes**, the time spent playing each each note should be decreased to **0.7s**; the delay in between playing the notes should be decreased to **0.25s**.
4. Once the sequence gets to **thirteen notes**, the time spent playing each each note should be decreased to **0.6s**; the delay in between playing the notes should be decreased to **0.15s**.

   Note that this should not affect the normal play and delay times when the player presses the switches!

(3) **Over time, no longer use the LEDs when playing the sequence.** Once the sequence gets to **fifteen notes**, stop turning LEDs on/off. That is, the player is evidently so good at this point that the sequence should only be played audibly. The play and delay times should be as described in (2) above (i.e., the same times as a sequence of thirteen notes).

Note that this should not affect the normal behavior of the LEDs when the player presses the switches!

You are to submit your Python source code only (as a .py file) through the upload facility on the web site.