

CSC/CYEN 132

The Science of Computing III

Living *with* Cyber

Student Edition

CSC/CYEN 132: The Science of Computing III

Living *with* Cyber (part 3 of 3)

Course Description: Additional coverage of algorithm analysis and development, object-oriented programming, data structures, computer architecture, and problem solving; applications of computing. This is the third 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. Understand more advanced data structures and their applications (e.g., dictionaries, lists, sets);
3. Be able to write object-oriented programs in a general-purpose programming language (e.g., Python);
4. Understand more advanced concepts of the object-oriented paradigm (e.g., polymorphism, coupling, and cohesion);
5. Have a basic understanding of signed and floating point number, and character representation in computers;
6. Have a basic understanding of how a computer is made (e.g., ALU, CPU, memory, I/O);
7. Have exposure to various applications of computing (e.g., problem solving with computers, software engineering, artificial intelligence); and
8. Work in groups on a significant computing project with a presentation.

Prerequisite(s): A grade of **C** or better in CSC 131 or CYEN 131.

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/project:	~30%
Programs:	~15%
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 on Data Structures
- More on Objects
- Building a Computer
- Application (Beam): Problem Solving with Computers
- Application (Beam): Software Engineering
- Algorithms...Reloaded
- Application (Beam): Artificial Intelligence

Lesson Summary

#	Title	Pillar(s)	Description/Topic(s)	Periods
01	More on Data Structures	Data Structures	<ol style="list-style-type: none"> 1. Useful list functions 2. List comprehensions 3. Sets 4. Dictionaries 5. Dictionary comprehensions 6. Room Adventure...Reloaded 7. Data structures summary 	2
02	More on Objects	Computer Programming	<ol style="list-style-type: none"> 1. Review of inheritance 2. Abstraction and modularization 3. Polymorphism and method lookup 4. Multiple inheritance 5. Abstract methods and abstract classes 6. Coupling and cohesion 	4
03	Building a Computer	Computer Architecture	<ol style="list-style-type: none"> 1. Signed numbers (signed magnitude, one's complement, two's complement) 2. Characters 3. Floating point numbers 4. Combinational circuits: decoders, encoders, multiplexers, and demultiplexers 5. Sequential circuits: R-S flip-flops and clocked R-S flip-flops 6. Building a simple computer 	5
04	Problem Solving with Computers	Beam (Application #2)	<ol style="list-style-type: none"> 1. The changing role of computers 2. Communication-oriented applications 3. Spreadsheets (including relative and absolute cell referencing, and replication and built-in functions) 4. Solving problems with spreadsheets 5. The Chaos Game spreadsheet 	1
05	Software Engineering	Beam (Application #3)	<ol style="list-style-type: none"> 1. Introduction to software engineering 2. Analysis and design 3. Documentation 4. Prototyping 5. Iterative software development 6. Errors, error handling, exceptions, exception handling, and error recovery 7. Testing and debugging 8. Software engineering in action: the game of life 	1
06	Algorithms...Reloaded	Algorithms, Computer	<ol style="list-style-type: none"> 1. Introduction to Java 2. A first Java program 	3

		Programming	3. Data types, constants, and variables 4. I/O, operators, and primary control constructs 5. Variable scope and program flow 6. Arrays 7. The foreach loop 8. The selection sort and binary search in Java 9. Laying out classes (Fraction...reloaded) 10. Inheritance, abstract classes and methods, and more (Shapes...reloaded)	
07	Artificial Intelligence	Beam (Application #4)	1. Introduction to artificial intelligence (AI) 2. Can intelligent machines be constructed? 3. The Turing Test 4. A brief history of AI 5. Game playing and search techniques 6. Automated reasoning 7. Neural networks and machine learning	1
Pi Activities				3
Final Pi Project				5
Exams				3
Housekeeping				1
Slack				1
TOTAL				30

CSC/CYEN 132: The Science of Computing III

Last updated:

13 Mar 2018

		Lessons	Raspberry Pi Activities	Puzzles	Videos	Assessments		
WEEK 1	1	Housekeeping						W
	2	More on Data Structures				Program 1 2D Points		F
	3			Eight Queens -- Knight's Tour	Microsoft HoloLens			M
WEEK 2	4	More on Objects						W
	5							F
	6					Program 2 2D Points...Plotted		M
WEEK 3	7							W
	8		Pi Activity 1 Room Adventure...Revolutions					F
	9		Final Pi Project (introduction)			Program 3 The Chaos Game		M
WEEK 4	10	Building a Computer						W
	11					Exam 1 More on Data Structures More on Objects		W
	12	Building a Computer		Nonogram -- Nonograms!	The Backwards Brain Bicycle			F
WEEK 5	13					Program 4 Shapes		M
	14							W
	15		Pi Activity 2 Room Adventure...Revolutions					F

CSC/CYEN 132: The Science of Computing III

Last updated:

13 Mar 2018

		Lessons	Raspberry Pi Activities	Puzzles	Videos	Assessments		
WEEK 6	16		Paper Piano					M
	17	Building a Computer						W
	18					Exam 2 Building a Computer		F
WEEK 7	19	Beam Problem Solving with Computers				Program 5 The Chaos Game...Reloaded		M
	20	Beam Software Engineering						W
	21		Final Pi Project	Einstein's Puzzle				F
WEEK 8	22	Algorithms...Reloaded						M
	23							W
	24		Final Pi Project					F
WEEK 9	25	Algorithms...Reloaded			The Expert			M
	26	Beam Artificial Intelligence			Marl/O: Machine Learning for Video Games -or- How Machines Learn			W
	27		Final Pi Project					F
WEEK 10	28					Exam 3 Algorithms...Reloaded Beams		M
	29		Final Pi Project (presentations)					W
	30	SLACK						F

Note: The beam on Software Engineering must be covered due to its material being required knowledge. Although the beam on Problem Solving with Computers is useful for students, the remaining beams can be replaced with others (based on instructor expertise and interest) as desired.

Lessons

So far, you have been introduced to various elementary and high level data structures: arrays, linked lists, stacks, queues, and binary trees. Specific to Python, you have used lists (similar to arrays) in your programs. In this lesson, we will discuss some powerful functions that work with lists and introduce several new data structures.

Useful list functions

As you have seen, Python lists are extremely useful data structures. In the first Python lesson, you were introduced to several list functions that, for example, reverse a list, sort a list, etc. In this lesson, we will cover several more powerful built-in functions that are quite useful when used with lists.

The `filter` function returns a new list consisting of only the items within an existing list for which some user-defined function is true. The user-defined function can be anything that evaluates an input in the existing list and returns true or false. The format for the `filter` function is as follows:

```
filter(function, list)
```

The parameter `function` represents the name of the function that will evaluate each item in the existing list. The parameter `list` is, of course, the existing list of items to evaluate.

Suppose, for example, that you want to find all of the multiples of three or five that are less than or equal to 30 and make a list of them. Here's one way to do this:

```
multiples = []

for i in range(3, 31):
    if (i % 3 == 0 or i % 5 == 0):
        multiples.append(i)
```

However, here's how it could be done with the `filter` function:

```
def f(x):
    return (x % 3 == 0 or x % 5 == 0)

multiples = filter(f, range(3, 31))
```

Both of these methods generate the following list:

```
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25, 27, 30]
```

The `map` function returns a new list consisting of the return values generated by a user-defined function on each item in an existing list. The user-defined function is called for each item in the existing list; the return values form the new list. The format for the `map` function is as follows:

```
map(function, list)
```

The parameters are the same as specified for the `filter` function. Suppose, for example, that you want to square each item in a list. Here's one way to do this:

```
squares = range(1, 10)
```

```

for i in range(len(squares)):
    squares[i] *= squares[i]

```

Although the snippet of code above does modify the existing list, it could be easily changed if needed. Here's how it could be done with the `map` function:

```

def f(x):
    return x * x

squares = map(f, range(1, 10))

```

Both methods produce the following list:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Lastly, the `reduce` function processes the elements in a list through a user-defined function and returns a *single* value. The function works by first processing the first two items of the list. The result of this is processed by the function, along with the next item in the list. This continues for all of the remaining items in the list. The format for the `reduce` function is as follows:

```
reduce(function, list)
```

Again, the parameters are the same as specified for the previous functions. Suppose, for example, that you want to compute the factorial of 10. Here's one way to do this:

```

fact = 1

for i in range(1, 11):
    fact *= i

```

Finally, here's how to do it with the `reduce` function:

```

def f(x, y):
    return x * y

fact = reduce(f, range(1, 11))

```

Both methods calculate the factorial of 10 (which is 3628800).

The following table summarizes the list functions discussed above:

Function	Purpose	Syntax	Returns
<code>filter</code>	Select list elements using a function	<code>filter(function, list)</code>	list
<code>map</code>	Apply a function to every list element	<code>map(function, list)</code>	list
<code>reduce</code>	Reduce a list to a single value using a function	<code>reduce(function, list)</code>	value

List comprehensions

Consider the simple problem of creating a list of the cubes of the integers 0, 1, 2, etc, up to 9 (i.e., 0, 1, 8, 27, 64, ..., 729). Try to write a snippet of Python code to accomplish this in the space below:

Another way uses a concept known as list comprehensions. A **list comprehension** provides a simple, concise way of creating lists (even complex ones). The most common use of this concept creates a list where each element is the result of some operation or expression applied to each element of another list. Here's an example that does the same thing as the snippet of code above:

```
cubes = [x * x * x for x in range(10)]
```

Yes, it's a single statement! A list comprehension uses the for loop to generate or to iterate through the items of a sequence and applies some operation or expression to each of those items. In the statement above, the generated sequence is the range of values from 0 through 9. The expression that is applied to each of the elements in the generated sequence is $x * x * x$ (i.e., it cubes each element). The result is a new list of the cubes of the elements in the generated sequence (0 through 9):

```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

The list comprehension can be read in English as, “the variable cubes is to be a list that contains some x cubed for each x in the range 0 through 9.” In fact, we can map the English version to the Python statement:

the variable cubes is to be	a list that contains	some value x cubed	for each x in the range 0 through 9
<code>cubes =</code>	<code>[</code>	<code>x * x * x</code>	<code>for x in range(10)]</code>

Minimally, a list comprehension consists of brackets containing an expression (e.g., $x * x * x$) followed by a for-loop. Additional for-loops or even if-statements can be chained after the first for-loop. The resulting list is an evaluation of the expression in the context of the for-loops and if-statements that follow it. Here's a seemingly convoluted example:

```
sums = [x+y for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```

The output of this statement is:

```
[4, 5, 5, 3, 6, 4, 7]
```

This statement generates a list of the sums of the pairs that can be formed by combining a single element from the first list (`[1, 2, 3]`) with a single element from the second list (`[3, 1, 4]`), **so long as the elements differ** (i.e., `if x != y`). The elements of each list are processed from left-to-right. The first sum is calculated by adding the first element of the first list to the first element of the second list ($1 + 3 = 4$). The second sum is calculated by adding the first element of the first list to the third element of the second list ($1 + 4 = 5$). Why the third element and not the second? Because this would mean that both elements have the same value (1). The if statement ensures that sums will only be produced if the list elements differ.

Here's another (similar) example of list comprehension:

```
pairs = [[x, y] for x in [1, 2, 3] for y in [3, 1, 4] if x < y]
```

The output of this statement is:

```
[[1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

This statement generates a list of the pairs that can be formed by combining a single element from the first list ([1, 2, 3]) with a single element from the second list ([3, 1, 4]), **so long as the element from the first list is less than the element from the second list**. Again, the list elements are processed from left-to-right. It is similar to the previous statement; however, instead of generating a list of sums, a list of pairs of elements is generated. In fact, it's a list of lists! The enclosed sublists are composed of only two elements each.

Really cool things can be done with list comprehension. For example, here's a neat way to compute pi to various precisions from two through eight digits:

```
from math import pi
```

```
pi = [round(pi, n) for n in range(2, 9)]
```

And here's the output:

```
[3.14, 3.142, 3.1416, 3.14159, 3.141593, 3.1415927, 3.14159265]
```

The function `round` does what you expect it to do: it *rounds* the value specified in the first parameter to a precision specified in the second parameter. For example, the expression `round(pi, 3)` rounds pi to three digits to the right of the decimal point (i.e., 3.142).

Sets

In Python, a set is just another type of sequence. The mathematical definition of a set is an unordered collection of unique elements. That is, a set is basically just a list with no duplicates. Since sets are mathematically defined, they support mathematical operations such as union, intersection, and difference. Defining a set can be done in several ways. The first is to formally define one, very much like you would define a list. However, instead of using square brackets, we use curly braces:

```
a = { 1, 2, 3, 4, 5 }
```

```
b = { 3, 4, 5, 6, 7 }
```

These Python statements declare and initialize two sets, *a* and *b*. Another way to create a set is to do so from some other sequence (such as a list):

```
c = [ 3, 1, 4, 1, 5, 9 ]
```

```
d = set(c)
```

The list *c* is used as input to create the set *d*. The set *d* will only have one instance of any duplicated element in the list *c*; therefore, the value of the set *d* is as follows:

```
{1, 3, 4, 5, 9}
```

Note that the element 1 appears only once in the set *d*. In addition, the set is unordered; that is, its elements don't necessarily have to be in the same order as those in the list.

A set can even be created from a string (since a string is a sequence):

```
e = set("sweet")
```

The value of the set *e* is as follows:

```
{ 's', 'e', 't', 'w' }
```

Again, note that the elements of this set are unique. To illustrate the set operations (union, intersection, and difference), we will use the sets *a* and *b* defined above. The **union** of the sets *a* and *b* represents the elements **in either *a* or *b***. The Python expression for this is written as *a* | *b*. Its output is the following set:

```
{1, 2, 3, 4, 5, 6, 7}
```

These elements are all of the unique elements in *a* or *b*. Similarly, the **intersection** of the sets *a* and *b* represents the elements **in both *a* and *b***. The Python expression for this is written as *a* & *b*. Its output is the following set:

```
{3, 4, 5}
```

These elements are the only unique elements in both *a* and *b*. Lastly, the **difference** of the sets *a* and *b* represents the elements **in *a* but not in *b***. The Python expression for this is written as *a* - *b*. Its output is the following set:

```
{1, 2}
```

The difference operation can be thought of as a subtraction of the set *b* from the set *a*. All elements in both *a* and *b* are removed. The remaining elements in *a* make up the resulting set. Any additional elements in *b* that are not in *a* are ignored.

Dictionaries

A dictionary is perhaps one of the most powerful data structures at our disposal in Python. As you have seen, sequences (like lists) are indexed by a range of numbers (i.e., the first element is placed at index 0, the second element is placed at index 1, and so on). A **dictionary** is a data structure whose elements are indexed by unique *keys*. A **key** is just an unchangeable value. The elements are known as **values**, and are associated with the keys. That is, a single key *maps* to a single value. This is why we often say that dictionaries contain key-value pairs.

Technically, a list pairs an index (which could be called a key) with a value (the element at that index). The difference is that dictionaries permit keys to be of almost any type, so long as a key is not susceptible to change. That is, it must be **immutable**. For example, the integer 5, the floating point number 3.14, the string “Jones”, and the character '%' are all valid keys. Note that all keys in a single dictionary must be unique (i.e., there can be no duplicate keys).

Consider a dictionary that you are familiar with: the kind that you lookup the definitions of words in. Using such a dictionary typically involves searching for some word in order to obtain its definition. In such a dictionary, the word is the key, and its definition is the value associated with that key. You should have noticed that, to search a dictionary, the key is required. The unknown is the value that is associated with the key (a definition). In some programming languages, this type of data structure is known as an **associative array**.

Another dictionary data structure that you are probably familiar with is a phone book (although you've probably only used some online version and not an actual book). What are the keys in a phone book? What about the values? Clearly, a name is the key (e.g., Bob Jones). The values associated with the keys are records that contain an address and a phone number. Certainly, such records can be represented as long strings (perhaps even with newlines). But we may also wish to represent the records as objects of some **PersonInfo** class!

Dictionaries are created similarly to sets (using braces). The difference is that key-value pairs are specified in the format `key: value`. Here's an example of a dictionary with strings representing last names as the keys and integers representing office numbers as the values:

```
offices = { "Jones": 247, "Smith": 121, "Kennedy": 108 }
```

This creates a dictionary with the following key-value pairs (in no particular order):

Last name	Office number
Jones	247
Smith	121
Kennedy	108

The main operations associated with a dictionary are to store some key-value pair and to retrieve a value associated with a key. Adding the new key-value pair `"Wilkerson": 355`, for example, can be added to the dictionary above as follows:

```
offices["Wilkerson"] = 355
```

The dictionary now has the following key-value pairs (in no particular order):

Last name	Office number
Jones	247
Smith	121
Kennedy	108
Wilkerson	355

Retrieving a value matching the key `"Smith"`, for example, can be done as follows:

```
loc = offices["Smith"]
```

The value of the variable `loc` is therefore 121. Note that attempting to retrieve a value using a key that is not in the dictionary results in an error.

An existing key-value pair in the dictionary may be overwritten by simply inserting a new value with the same key. For example, suppose that Kennedy changed offices (to, say, 111). The dictionary can be updated as follows:

```
offices["Kennedy"] = 111
```

The dictionary now has the following key-value pairs (in no particular order):

Last name	Office number
Jones	247
Smith	121
Kennedy	111
Wilkerson	355

A key-value pair can be removed from the dictionary using the `del` keyword as follows:

```
del offices["Smith"]
```

The dictionary now has the following key-value pairs (in no particular order):

Last name	Office number
Jones	247
Kennedy	111
Wilkerson	355

Determining if a key is in the dictionary without actually returning the value associated with the key can be done by using the keyword `in` as follows:

```
"Kennedy" in offices → true
"Smith" in offices → false (since it was just removed)
if ("Smith" in offices):
    ...
```

The keys and values in a dictionary don't have to be homogeneous; that is, they can each be of different types. For instance, the following key-value pair could be added to the dictionary:

```
offices[12345] = "abracadabra"
```

Although it doesn't necessarily make sense, the dictionary now has the following key-value pairs (in no particular order):

Last name	Office number
Jones	247
Kennedy	111
Wilkerson	355
12345	abracadabra

A neat way to obtain a list of all of the keys in a dictionary is to use the `keys` function as follows:

```
keys = offices.keys()
```

The list `keys` then has the following value:

```
['Jones', 12345, 'Wilkerson', 'Kennedy']
```

There are several ways of iterating through a dictionary. One uses the `keys` function just described. This can be accomplished as follows:

```
for v in offices.keys():
    print offices[v]
```

The output of this snippet of Python code is:

```
247
abracadabra
355
111
```

Of course, to produce a key-value pair mapping, only a small modification is required:

```
for v in offices.keys():
    print v, "->", offices[v]
```

The output of this now includes both the keys and values:

```
Jones -> 247
12345 -> abracadabra
Wilkerson -> 355
Kennedy -> 111
```

Another way that Python provides to do the same thing and which produces the same output is to use the dictionary method `iteritems` as follows:

```
for k,v in offices.iteritems():
    print k, "->", v
```

The `iteritems` function returns pairs of values, each of which is a key-value pair in the dictionary.

Dictionary comprehensions

Just as with lists, comprehensions can be used to create dictionaries. Of course, these are known as dictionary comprehensions. Here's one that creates a dictionary with the key-value pairs such that the values are cubes of the keys, and the keys range from 1 through 5:

```
dict = {x: x**3 for x in range(1, 6)}
```

The created dictionary `dict` is therefore `{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}`. The key-value pairs are specified in the dictionary comprehension as `x: x**3` (i.e., a key is some value `x`, and its associated value is `x` cubed). The range for the values (1 through 5) taken on by the variable `x` is specified as `for x in range(1, 6)`.

In the space below, try to modify the dictionary comprehension above so that the values are stored as strings instead of integers:

One method is as follows:

```
dict = {x: str(x**3) for x in range(1, 6)}
```

The dictionary is now {1: '1', 2: '8', 3: '27', 4: '64', 5: '125'}. The difference is subtle; however, the values are now strings instead of integers.

Activity 1: Room Adventure...Reloaded

In this activity, we will update the Room Adventure game that was designed in a previous RPi activity. The goal will be to replace the parallel arrays in the game with dictionaries. Such a substitution makes sense because parallel arrays associate (or map) the elements of two or more arrays by index value. That is, the first element of one array is paired with the first element of another, and so on. Dictionaries are perfectly suited for this because they associate one value with another!

Recall that parallel arrays were used to represent the following relationships:

- (1) Exits with exit locations (through the lists `exits` and `exitLocations`); and
- (2) Items with item descriptions (through the lists `items` and `itemDescriptions`).

Exits were strings like “north” and “west”, and exit locations were rooms (instances of the class **Room**). Items were strings like “table” and “fireplace”, and item descriptions were strings like “It is made of oak. A golden key rests on it.” and “It is full of ashes.”

Very quickly, we see that we can replace the lists `exits` and `exitLocations` with a single dictionary (perhaps just called `exits`). Suppose, for example, that an exit to the east led to some instance of a room represented by the variable `r2`, and an exit to the north led to some instance of a room represented by the variable `r3`. A dictionary that represents this could be created as follows:

```
exits = {"east": r2, "north": r3}
```

Of course, this supposes that the variables `r2` and `r3` exist.

Step 1: Replace the parallel lists with dictionaries

The first thing to do to modify our game is to remove the parallel arrays for both exits and items, and replace them with dictionaries. This must be done in the constructor of the **Room** class; specifically, in lines 16 through 19:

```
14: def __init__(self, name):
15:     self.name = name
16:     self.exits = []
17:     self.exitLocations = []
18:     self.items = []
19:     self.itemDescriptions = []
20:     self.grabbables = []
```

A few notes: (1) line numbers specified in this activity are valid only within the existing source code (i.e., not the one that is being modified because changes may invalidate the line numbers); and (2) some comments in the source code have been removed in this activity for brevity. The statements on lines 16

through 19 are the parallel arrays that will need to be replaced with dictionaries. The lists `exits` and `exitLocations` are paired; so are the lists `items` and `itemDescriptions`. Let's replace them with two dictionaries instead:

```
def __init__(self, name):
    self.name = name
    self.exits = {}
    self.items = {}
    self.grabbables = []
```

The highlighted statements are the two new dictionaries. There is no longer a need for matching lists since the dictionaries intrinsically match keys to values! Pay attention to the braces (as opposed to brackets).

Step 2: Remove the accessors and mutators for the deleted parallel lists

Recall that accessors and mutators were implemented for each of a **Room**'s instance variables. Since the lists `exitLocations` and `itemDescriptions` were removed, their respective accessors and mutators must also be removed. The existing accessors and mutators for the instance variables `exits` and `items` remain unchanged. In fact, they will work seamlessly with the new dictionaries.

The accessor and mutator for the old instance variable `exitLocations` are located on lines 39 through 45, and must be removed from the source code:

```
39: @property
40: def exitLocations(self):
41:     return self._exitLocations
42:
43: @exitLocations.setter
44: def exitLocations(self, value):
45:     self._exitLocations = value
```

The accessor and mutator for the old instance variable `itemDescriptions` are located on lines 55 through 61, and must also be removed from the source code:

```
55: @property
56: def itemDescriptions(self):
57:     return self._itemDescriptions
58:
59: @itemDescriptions.setter
60: def itemDescriptions(self, value):
61:     self._itemDescriptions = value
```

Step 3: Modify the `addExit` and `addItem` functions

The next step is to change the `addExit` and `addItem` functions in the **Room** class so that they appropriately insert new exits and items into dictionaries instead of parallel lists as is currently done. First, let's change the `addExit` function, which begins on line 74:

```

74: def addExit(self, exit, room):
75:     # append the exit and room to the appropriate lists
76:     self._exits.append(exit)
77:     self._exitLocations.append(room)

```

Note how the function currently appends the exit to one list and the room to another. Let's change this so that the exit and room are added as a key-value pair to the appropriate dictionary:

```

def addExit(self, exit, room):
    # append the exit and room to the appropriate dictionary
    self._exits[exit] = room

```

Now, let's change the addItem function, which begins on line 82:

```

82: def addItem(self, item, desc):
83:     # append the item and description to the appropriate lists
84:     self._items.append(item)
85:     self._itemDescriptions.append(desc)

```

This function is similar to the old addExit function. It appends the item to one list and the description to another. Let's change this so that the item and description are added as a key-value pair to the appropriate dictionary:

```

def addItem(self, item, desc):
    # append the item and description to the appropriate dictionary
    self._items[item] = desc

```

Step 4: Modify the `__str__` function

In the game, the player is continually presented with the status: location, items, exits, and inventory. This is specified in the **Room** class and specifically involves the bit of source code that generates the string representation of a **Room**. This is done in the `__str__` function, which begins on line 100:

```

100: def __str__(self):
101:     # first, the room name
102:     s = "You are in {}. \n".format(self.name)
103:
104:     # next, the items in the room
105:     s += "You see: "
106:     for item in self.items:
107:         s += item + " "
108:     s += "\n"
109:
110:     # next, the exits from the room
111:     s += "Exits: "
112:     for exit in self.exits:
113:         s += exit + " "
114:
115:     return s

```

The statements that need to be changed are on lines 106 and 112. In their current form, they iterate through the two lists. The list `items` used to contain the items (e.g., table). Since `items` is now a

dictionary (with the items as keys and item descriptions as values), then we must iterate through its keys! This can be done by replacing line 106 with the following statement:

```
for item in self.items.keys():
```

Similarly, The list `exits` used to contain the exits (e.g., south). Since `exits` is now a dictionary (with the exits as keys and room objects as values), then we must also iterate through its keys. This can be done by replacing line 112 with the following statement:

```
for exit in self.exits.keys():
```

Step 5: Modify the main part of the program

There are two places that need modification in the main part of the program. Both are required because the current source code refers to the old lists that have been replaced by dictionaries. The first modification occurs in the part of the source code that is executed if the verb in the action specified by the player is *go* (e.g., “go south”). This part of the source code begins on line 247. Note that some of the next part of the if statement is provided for clarity:

```
247: if (verb == "go"):
248:     # set a default response
249:     response = "Invalid exit."
250:
251:     # check for valid exits in the current room
252:     for i in range(len(currentRoom.exits)):
253:         # a valid exit is found
254:         if (noun == currentRoom.exits[i]):
255:             # change the current room to the one ...
256:             currentRoom = currentRoom.exitLocations[i]
257:             # set the response (success)
258:             response = "Room changed."
259:             # no need to check any more exits
260:             break
261: # the verb is: look
262: elif (verb == "look"):
263:     ...
```

Currently, the algorithm iterates through the list of exits. If one matches the noun specified by the player, then the current room is changed to the matching exit location (in the parallel list). If this occurs, the `break` statement exits the loop (i.e., we don't need to check for more exits since a valid one has already been found). The changes required affect the highlighted lines in the snippet of source code above. Ultimately, the part of the if statement that is executed if the verb is *go* should be changed to the following source code. Again, some of the next part of the if statement is provided for clarity:

```
# the verb is: go
if (verb == "go"):
    # set a default response
    response = "Invalid exit."

    # check for valid exits in the current room
    if (noun in currentRoom.exits):
        # if one is found, change the current room ...
        currentRoom = currentRoom.exits[noun]
```

```

        # set the response (success)
        response = "Room changed."
# the verb is: look
elif (verb == "look"):
    ...

```

Note that no break statement is required because there is no enclosing loop! The second modification occurs in the part of the source code that is executed if the verb in the action specified by the player is *look* (e.g., “look table”). This part of the source code begins on line 261. Again, some of the next part of the if statement is provided for clarity:

```

261: # the verb is: look
262: elif (verb == "look"):
263:     # set a default response
264:     response = "I don't see that item."
265:
266:     # check for valid items in the current room
267:     for i in range(len(currentRoom.items)):
268:         # a valid item is found
269:         if (noun == currentRoom.items[i]):
270:             # set the response to the item's description
271:             response = currentRoom.itemDescriptions[i]
272:             # no need to check any more items
273:             break
274: # the verb is: take
275: elif (verb == "take"):

```

As before, the algorithm iterates through a list (of items in this case). If one matches the noun specified by the player, then the response is changed to the matching item description (in the parallel list). If this occurs, the break statement exits the loop (i.e., we don't need to check for more items since a valid one has already been found). The changes required affect the highlighted lines in the snippet of source code above. Ultimately, the part of the if statement that is executed if the verb is *look* should be changed to the following source code. Again, some of the next part of the if statement is provided for clarity:

```

# 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
    if (noun in currentRoom.items):
        # if one is found, set the response to the ...
        response = currentRoom.items[noun]
# the verb is: take
elif (verb == "take"):

```

And that's it!

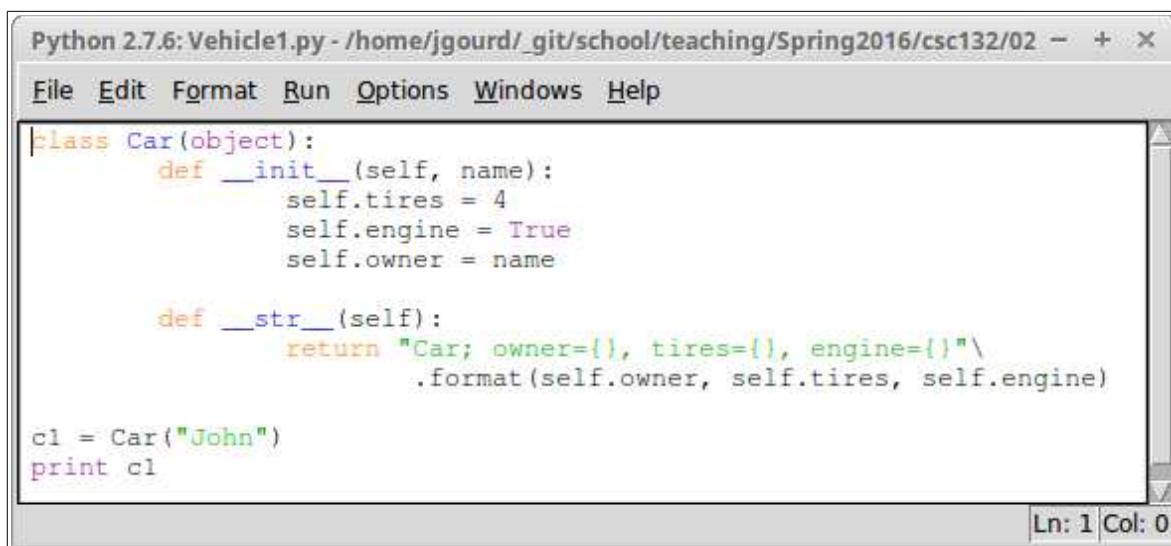
Wrapping up data structures

By now, you should have a good idea of the need for data structures. Generally, programs that solve problems store and manipulate data in some way. Data structures make this possible. In this curriculum, various useful data structures were introduced. Here is a table that summarizes them:

Data structure	Description
Array	Similar pieces of data store in contiguous memory locations. Data values (elements) are stored at index locations. The size of an array must be known before using one.
Linked list	Similar pieces of data stored in nodes located in various memory locations. Nodes store data values and a link to the next node in memory. The list has a head (the first element) and a tail (the last element). The size of a linked list can grow or shrink as needed.
Stack	A list-like structure where inserting and deleting is performed at one end (usually called the top of the stack). Inserting is called a push operation; deleting is called a pop operation. The last item inserted in a stack is the first item out of the stack; therefore, a stack is a LIFO (last-in, first-out) data structure.
Queue	A list-like structure where inserting is performed at one end (usually called the back of the queue) and deleting is performed at the other end (usually called the front of the queue). Inserting is called an enqueue operation; deleting is called a dequeue operation. The first item inserted in a queue is the first item out of the queue; therefore, a queue is FIFO (first-in, first-out) data structure.
Binary tree	A tree-like data structure made up of nodes that store similar pieces of data. Each node has links to (up to) two children. The binary tree has a root (the top node) and leaves (nodes at the bottom with no children). From any node, a subtree can be described such that the node is the root of that subtree.
Ordered binary tree	A binary tree such that the values of all children in the left subtree of each node are less than the value of the node, and the values of all children in the right subtree of each node are greater than or equal to the value of the node.
Dictionary	Also known as an associative array, a data structure that maps keys to values. Keys are unchangeable pieces of data; values are associated with keys (one value per key).

The last major topic that was discussed in the object-oriented paradigm was inheritance. To review, there are cases where it is advantageous to have one class (called a subclass) inherit state and behavior from another class (called a superclass). Using inheritance allows us, among many other things, to avoid excessive repetition and code duplication. This makes maintaining code much easier.

As an example, let's assume that you are writing a program to store information about the kinds of vehicles that a car mechanic's garage may deal with on a daily basis. For simplicity, suppose that the mechanic only keeps track of the number of tires that a car has (yes, there are some cars that have three tires¹), whether the car has a working engine, and the car's owner. In this case, the car class would need three instance variables (one for each of the mentioned attributes). It may even have a function that allows a car's information to be output in a meaningful way (i.e., it would implement a `__str__` method). Even though the class is very basic, we'll use it to make a point. Here is the source code for the `Car` class, along with some sample testing code in the main part of the program:



```
Python 2.7.6: Vehicle1.py - /home/jgourd/_git/school/teaching/Spring2016/csc132/02 - + x
File Edit Format Run Options Windows Help

class Car(object):
    def __init__(self, name):
        self.tires = 4
        self.engine = True
        self.owner = name

    def __str__(self):
        return "Car; owner={}, tires={}, engine={}"\
            .format(self.owner, self.tires, self.engine)

c1 = Car("John")
print c1

Ln: 1 Col: 0
```

Of course, the output is fairly simple to determine:

```
Car; owner=John, tires=4, engine=True
```

¹ Check out the Reliant Robin. In particular, watch the Top Gear episode in which Jeremy Clarkson drives one...with a comedic effect.

Now consider the case where, one day, the mechanic's garage decides to repair bicycles as well. A very quick fix would be to create a similar class just for bicycles. In fact, here are comparisons of the **Car** and **Bicycle** classes (with a slightly modified main part of the program):

A screenshot of a Python 2.7.6 IDE window. The title bar reads "Python 2.7.6: trial.py - /home/jgourd/_git/school/teaching/Spring2016/csc132/02 M...". The menu bar includes "File", "Edit", "Format", "Run", "Options", "Windows", and "Help". The code editor contains the following Python code:

```
class Car(object):
    def __init__(self, name):
        self.tires = 4
        self.engine = True
        self.owner = name

    def __str__(self):
        return "Car; owner={}, tires={}, engine={}"\
            .format(self.owner, self.tires, self.engine)

class Bicycle:
    def __init__(self, name):
        self.tires = 2
        self.engine = False
        self.owner = name

    def __str__(self):
        return "Bicycle; owner={}, tires={}, engine={}"\
            .format(self.owner, self.tires, self.engine)

c1 = Car("John")
b1 = Bicycle("Jane")
print c1
print b1
```

The status bar at the bottom right shows "Ln: 1 Col: 0".

And here is the output of this modified program:

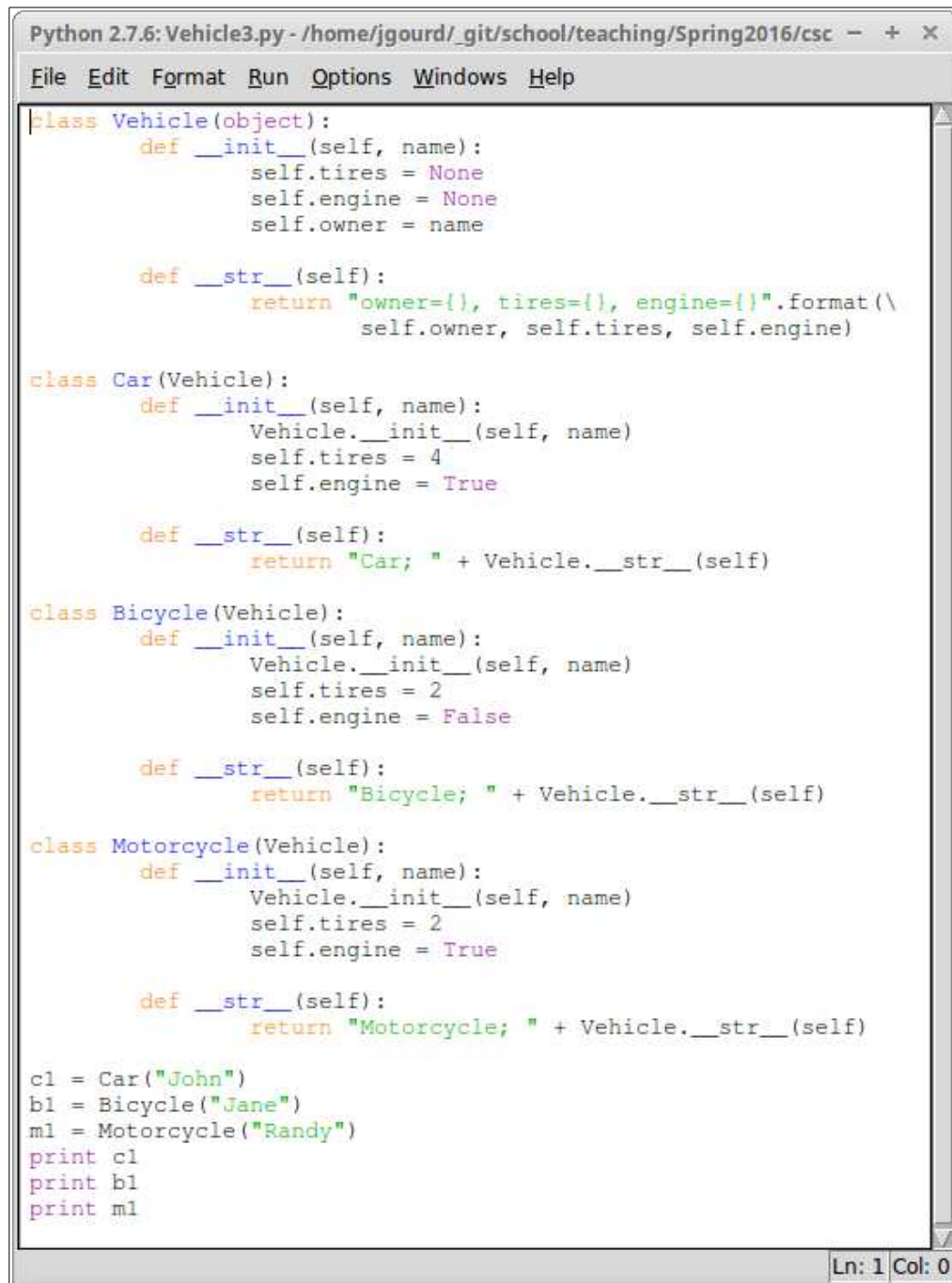
```
Car; owner=John, tires=4, engine=True
Bicycle; owner=Jane, tires=2, engine=False
```

Notice the code duplication? In fact, the **Bicycle** class is almost an exact copy of the **Car** class. The only differences are that a bicycle has only two tires, has no engine, and is called a bicycle. As you might imagine, creating similar classes this way gets very ineffective the larger classes get, and the more classes an application requires. For example, what would need to be done if the garage decided to repair motorcycles as well? Expanding the example above, another class would have to be created. In fact, it would also be very similar to the **Car** class.

What would happen if a mistake were made in the source code that implements the way that tires are accounted for? Or suppose that an adjustment needed to be made (e.g., adding another instance variable to keep track of a spare tire). In the best case, modifications to all three classes would have to be made, thereby increasing the chances of making mistakes and causing errors.

As discussed previously, the concept of inheritance allows us to create a superclass that can encapsulate all of the common members that one or more subclasses can share. The subclasses that are created from some superclass can be considered to be specific subtypes of the superclass.

One way to modify the code in the example above to utilize inheritance is to create a **Vehicle** superclass that defines all of the members that are shared among all types of vehicles in the application: cars, bicycle, and motorcycles. The car, bicycle, and motorcycle classes would then inherit from and become subclasses of the **Vehicle** class. This greatly helps to reduce code duplication. To illustrate this, here are modified classes, along with some test code, that updates the classes above and implements inheritance:



```
Python 2.7.6: Vehicle3.py - /home/jgourd/.git/school/teaching/Spring2016/csc - + x
File Edit Format Run Options Windows Help

class Vehicle(object):
    def __init__(self, name):
        self.tires = None
        self.engine = None
        self.owner = name

    def __str__(self):
        return "owner={}, tires={}, engine={}".format(\
            self.owner, self.tires, self.engine)

class Car(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.tires = 4
        self.engine = True

    def __str__(self):
        return "Car; " + Vehicle.__str__(self)

class Bicycle(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.tires = 2
        self.engine = False

    def __str__(self):
        return "Bicycle; " + Vehicle.__str__(self)

class Motorcycle(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.tires = 2
        self.engine = True

    def __str__(self):
        return "Motorcycle; " + Vehicle.__str__(self)

c1 = Car("John")
b1 = Bicycle("Jane")
m1 = Motorcycle("Randy")
print c1
print b1
print m1

Ln: 1 Col: 0
```

And here is the output of the program:

```
Car; owner=John, tires=4, engine=True
Bicycle; owner=Jane, tires=2, engine=False
Motorcycle; owner=Randy, tires=2, engine=True
```

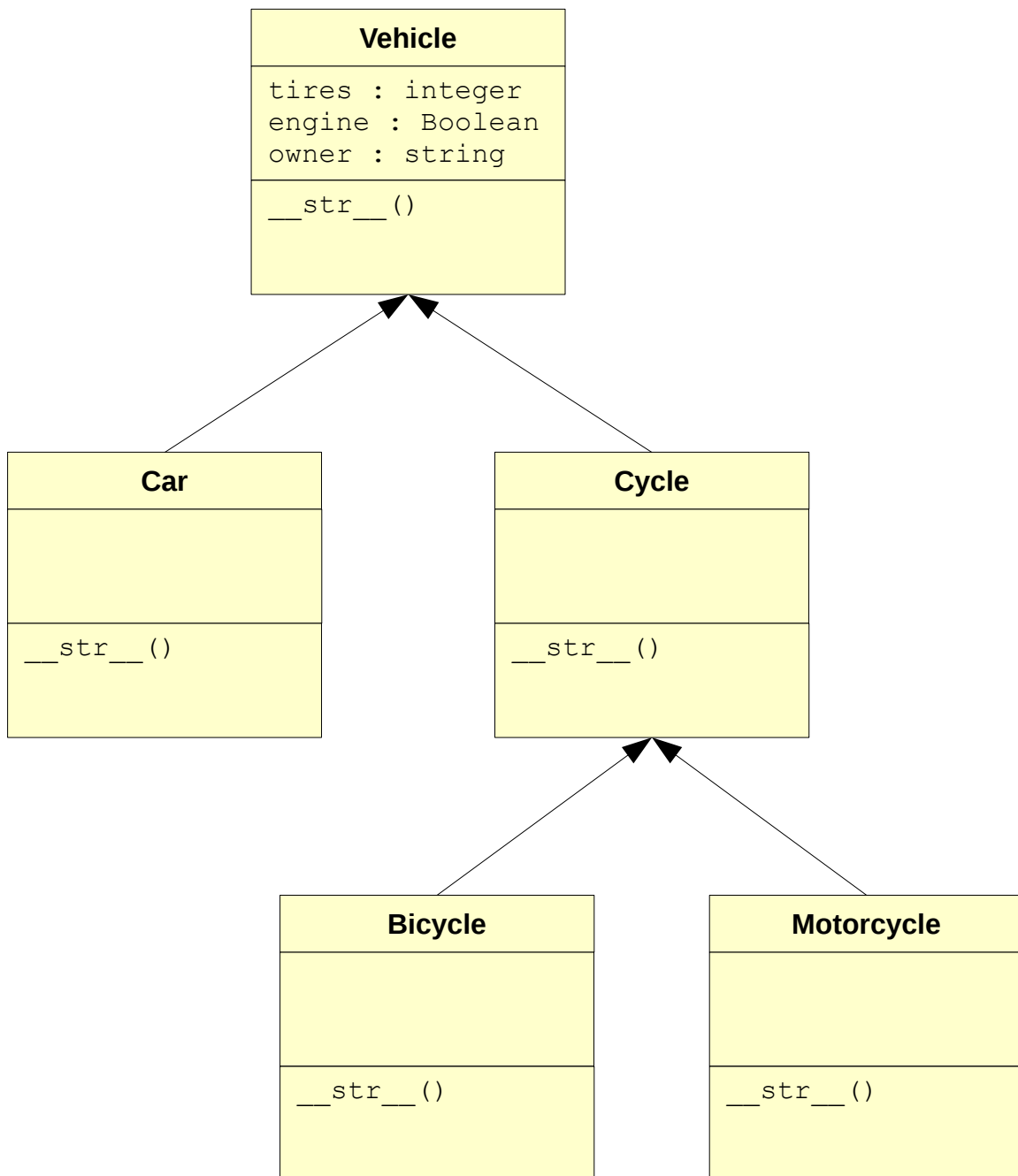
Look over the code above and see if it makes sense to you. Notice the default values of `None` for the variables `tires` and `engine`. These values are overwritten in the subclasses, depending on which subclass is initialized, as follows:

Class	tires	engine
Vehicle	None	None
Car	4	True
Bicycle	2	False
Motorcycle	2	True

Also notice that the constructor of the **Vehicle** class (the superclass) is called in the constructor of each of the subclasses. In fact, this is the first statement in the constructors of the subclasses. That is, the initialization of a car, bicycle, and motorcycle first means to initialize a vehicle. This sets default values for the variables `tires` and `engine`. The owner of the vehicle is passed in when instantiating each of the subclasses, and is forwarded to the superclass (where the value is formally assigned). Any remaining initialization of the subclasses is done after the call to the constructor in the superclass (i.e., specific values for the variables `tires` and `engine`).

Also, the class output magic function (`__str__`) also calls the matching function in the **Vehicle** superclass. Why? Well, it actually generates the appropriate output for any kind of vehicle (i.e., owner, number of tires, and whether or not it has an engine). The only distinguishing characteristic is the actual name of the class. Therefore, the subclasses first generate a string matching their name followed by a call to the superclass function (which generates a string containing the information specified above).

As a last modification, note that bicycles and motorcycles have the same number of tires. Since we can have many levels of inheritance, let's try to add a generic **Cycle** class that encapsulates the attributes shared by all kinds of *cycles* (i.e., bicycles and motorcycles). To be clear, the application now has the following class diagram:



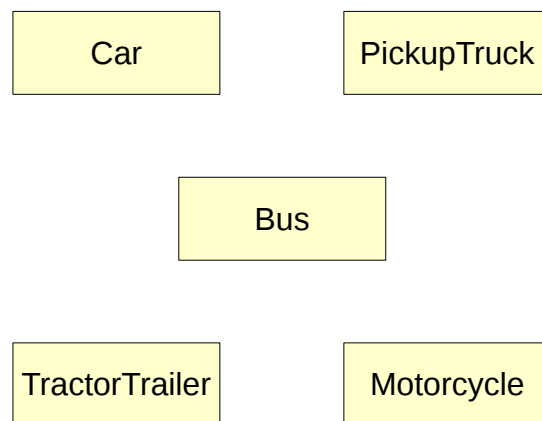
In the space below, implement the cycle class; then, modify the bicycle and motorcycle classes:

The act of designing an application, particularly a large one, requires consideration of many factors that help to ensure its success. That is, there are ideas that must be considered in order to make designing an application easier. Much of the remainder of this lesson is dedicated to identifying such techniques.

Abstraction and modularization

Abstraction is the ability to ignore the details of parts of a system in order to focus our attention at a *higher* level. Think of a Google map. When zoomed out to its default level, an entire city may be visible. Detailed street names, building structures, and so on, are invisible as they would clutter the map. Similarly, areas surrounding the city (e.g., state, region, country) are not visible. Zooming in reveals more detail of a part of the city; however, some of the surrounding detail that was present before is lost. Zooming out may further hide details, but bring in surrounding cities and towns. These *zoom levels* present just enough information that is required to process the map at that level.

The object-oriented paradigm is actually emblematic of this concept. Consider the traffic simulation application that was discussed in a previous lesson. In case you have forgotten, its goal was to model 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. As discussed, 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 the city. We initially modeled it with the following class diagram:



In the simulation, the way in which cars, pickup trucks, buses, motorcycles, and so on, are implemented doesn't matter to a city official using it to make zoning decisions. Those details are *abstracted* away from the user. As another example, the programmers tasked with extending the traffic simulation don't necessarily need to know how a bus works to, say, support school zones in the simulation.

At its most basic level, the concept of an object represents a way of abstracting away data and operations into a single *thing*, the data being state and operations being behavior. To instantiate an object and use it in some programming context, there is no real need to know how some behavior is actually implemented, for example. Simply understanding the interface (i.e., how to invoke some sort of behavior) is enough. We just need to know what function to call, what parameters to pass it, and if we should expect a return value.

Modularization is the act of dividing a whole into well-defined parts that can be built and examined separately. It is important to note, however, that the parts typically interact and must do so in well-

defined ways. This facilitates reasoning about and maintenance of the application. In the traffic simulation example, the act of designing various classes to best represent the components of the application inherently demonstrates modularization.

Often, abstraction and modularization go hand-in-hand. In a sense, modularization results in different levels of abstraction throughout some application. Moreover, the goal of setting levels of abstraction in an application (for example, to make maintenance easier and more manageable) motivates modularization.

Polymorphism

Let's go back to the mechanic's garage problem that was used earlier. Notice that all of the classes involved have their own `__str__` function. When we execute a statement such as `print b1`, which specific `__str__` function is executed? This problem is referred to as **method lookup**, and the idea of having multiple functions with the same name in multiple classes is called **polymorphism**.

At first, it may seem confusing to have multiple functions in different classes with the same name; however, it turns out to be a very handy feature of object-oriented programming. Let's answer the question that was initially posed: when we execute a statement such as `print b1`, which specific `__str__` function is executed? The variable `b1` is a bicycle. Therefore, if an `__str__` function exists in the **Bicycle** class, then it is executed. Indeed, one exists in the class. In fact, it contains the following single statement:

```
return "Bicycle; " + Cycle.__str__(self)
```

Note that the function also calls the `__str__` function in the **Cycle** class via the right-hand part of the statement: `Cycle.__str__(self)`. The entire string cannot be returned until `__str__` in the **Cycle** class has finished execution. It, too, contains only a single statement:

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

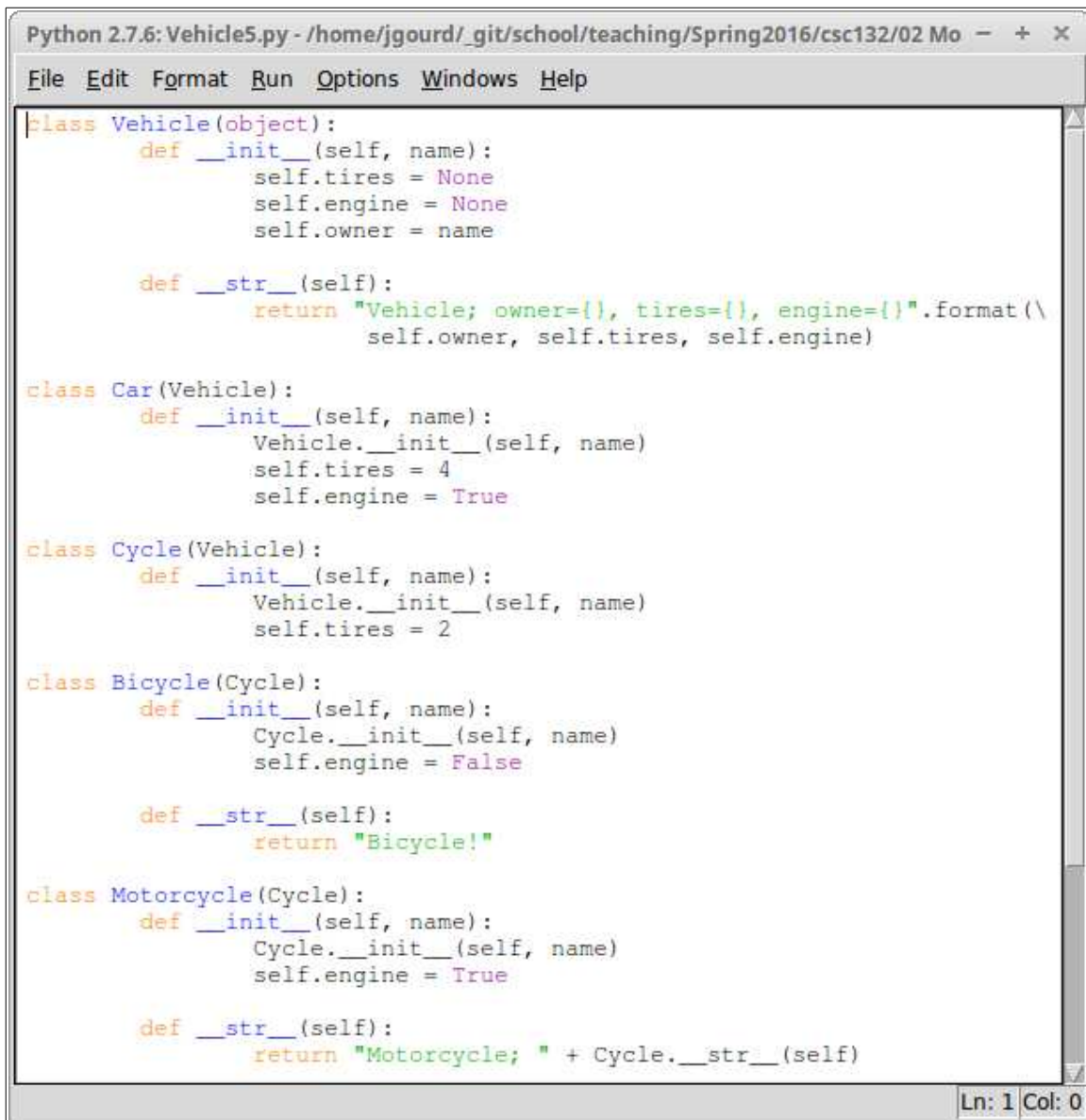
Clearly, it calls the `__str__` function in the **Vehicle** class, which returns a string containing the owner of the vehicle, its number of tires, and whether it has an engine or not:

```
return "owner={}, tires={}, engine={}".format(\
    self.owner, self.tires, self.engine)
```

In the end, all three `__str__` output functions are executed with the statement `print b1`. What about the statement `print c1`? In a similar manner, we see that the `__str__` function in the **Car** class is executed first (since `c1` is an instance of the **Car** class). This function calls its matching function in the **Vehicle** class (in order to actually produce the string containing the car's owner, number of tires, and whether it has an engine or not).

In both of these examples, the function that is called is the one located in the object reference's defining class. That is, since `b1` is a bicycle, then the function in the **Bicycle** class is called. Similarly, since `c1` is a car, then the function in the **Car** class is called. The fact that there may be chained calls to matching functions in superclasses is just coincidence (however, it is intentional in order to produce the proper output).

Let's take a look at a slightly modified version of the previous example. First, the classes:



```
Python 2.7.6: Vehicle5.py - /home/jgourd/_git/school/teaching/Spring2016/csc132/02 Mo - + x
File Edit Format Run Options Windows Help

class Vehicle(object):
    def __init__(self, name):
        self.tires = None
        self.engine = None
        self.owner = name

    def __str__(self):
        return "Vehicle; owner={}, tires={}, engine={}".format(\
            self.owner, self.tires, self.engine)

class Car(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.tires = 4
        self.engine = True

class Cycle(Vehicle):
    def __init__(self, name):
        Vehicle.__init__(self, name)
        self.tires = 2

class Bicycle(Cycle):
    def __init__(self, name):
        Cycle.__init__(self, name)
        self.engine = False

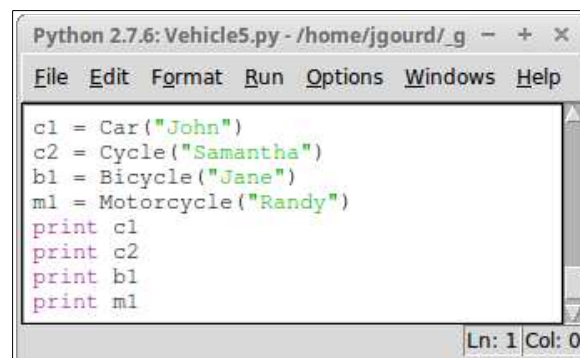
    def __str__(self):
        return "Bicycle!"

class Motorcycle(Cycle):
    def __init__(self, name):
        Cycle.__init__(self, name)
        self.engine = True

    def __str__(self):
        return "Motorcycle; " + Cycle.__str__(self)
```

Ln: 1 Col: 0

And the main part of the program:



```
Python 2.7.6: Vehicle5.py - /home/jgourd/_g - + x
File Edit Format Run Options Windows Help

c1 = Car("John")
c2 = Cycle("Samantha")
b1 = Bicycle("Jane")
m1 = Motorcycle("Randy")
print c1
print c2
print b1
print m1
```

Ln: 1 Col: 0

This example has the same classes as the previous one. However, the `__str__` functions have been changed in various ways. Given the discussion above about method lookup, can you explain what the print statements in the main part of the program will produce? Try to do so in the space below:

Method lookup is essentially a simple concept. If a specified function is not found in a class, then a search for the matching function is performed in the superclass. In fact, method lookup works by continuously trying to find a matching function in the superclass hierarchy until one is found. If one is not found, then an error occurs.

Polymorphism is a powerful concept. It allows us to specify a function at a superclass level (including any desired implementation), and then to overwrite it at lower levels of the class hierarchy (i.e., in subclasses). Consider this a way to specialize or refine behaviors defined in superclasses.

Let's look at an example of how this can be leveraged to produce efficient programs. Consider a scenario in which you want to write a program that can draw basic shapes (e.g., a rectangle). Such a program can be designed by creating a rectangle class that stores its `length` and `width` in instance variables, and has a `draw` function that produces a representation of that shape (for now, just using characters that can be found on a keyboard). A simple class diagram for this could be the following:

Rectangle
length : integer width : integer
draw()

And here's one possible method of implementing the class:

```

Python 2.7.6: Rectangle.py - /home/jgourd/_git/school/te...
File Edit Format Run Options Windows Help
class Rectangle(object):
    def __init__(self, l, w):
        self.length = l
        self.width = w

    def draw(self):
        for i in range(self.width):
            print "*" * self.length

r1 = Rectangle(10, 4)
r1.draw()
Ln: 1 Col: 0

```

Lastly, here's the output produced by the program above:

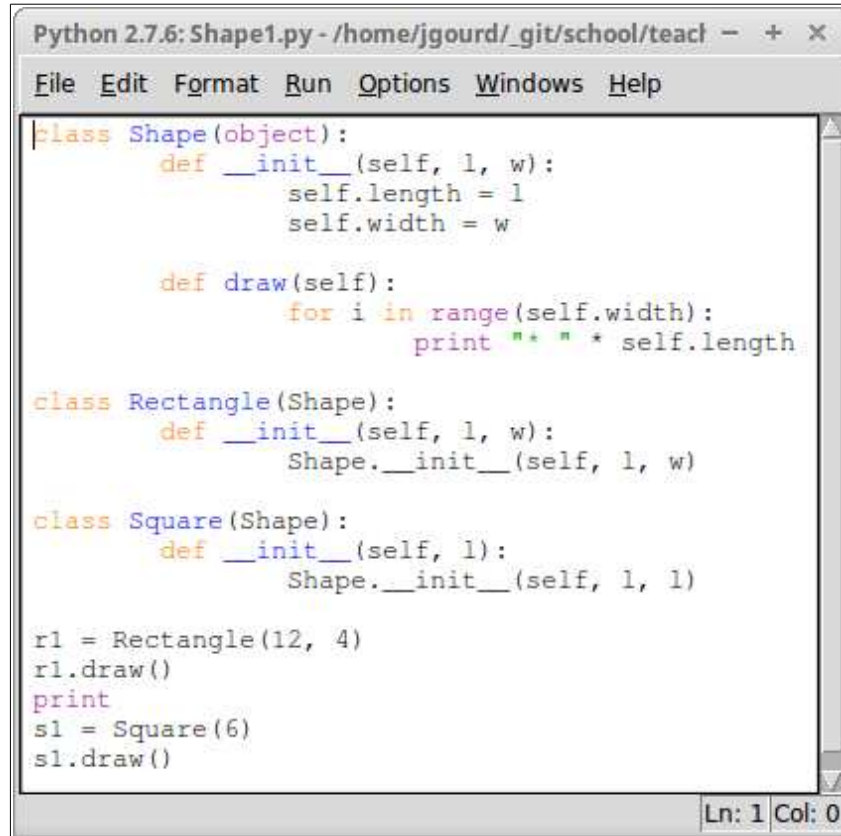
```

* * * * *
* * * * *
* * * * *
* * * * *

```

Sure enough, we asked for a rectangle that is 10 units long by 4 units high. And that's what we got! Now suppose that we want to expand the program to be able to accommodate squares. Squares are very similar to rectangles except that their length and width are always equal. Considering what we now know about inheritance and polymorphism, we could create a generic shape class that both squares and rectangles could inherit from.

Here's an updated version that implements this:



```
Python 2.7.6: Shape1.py - /home/jgourd/_git/school/teach - + x
File Edit Format Run Options Windows Help

class Shape(object):
    def __init__(self, l, w):
        self.length = l
        self.width = w

    def draw(self):
        for i in range(self.width):
            print "*" * self.length

class Rectangle(Shape):
    def __init__(self, l, w):
        Shape.__init__(self, l, w)

class Square(Shape):
    def __init__(self, l):
        Shape.__init__(self, l, l)

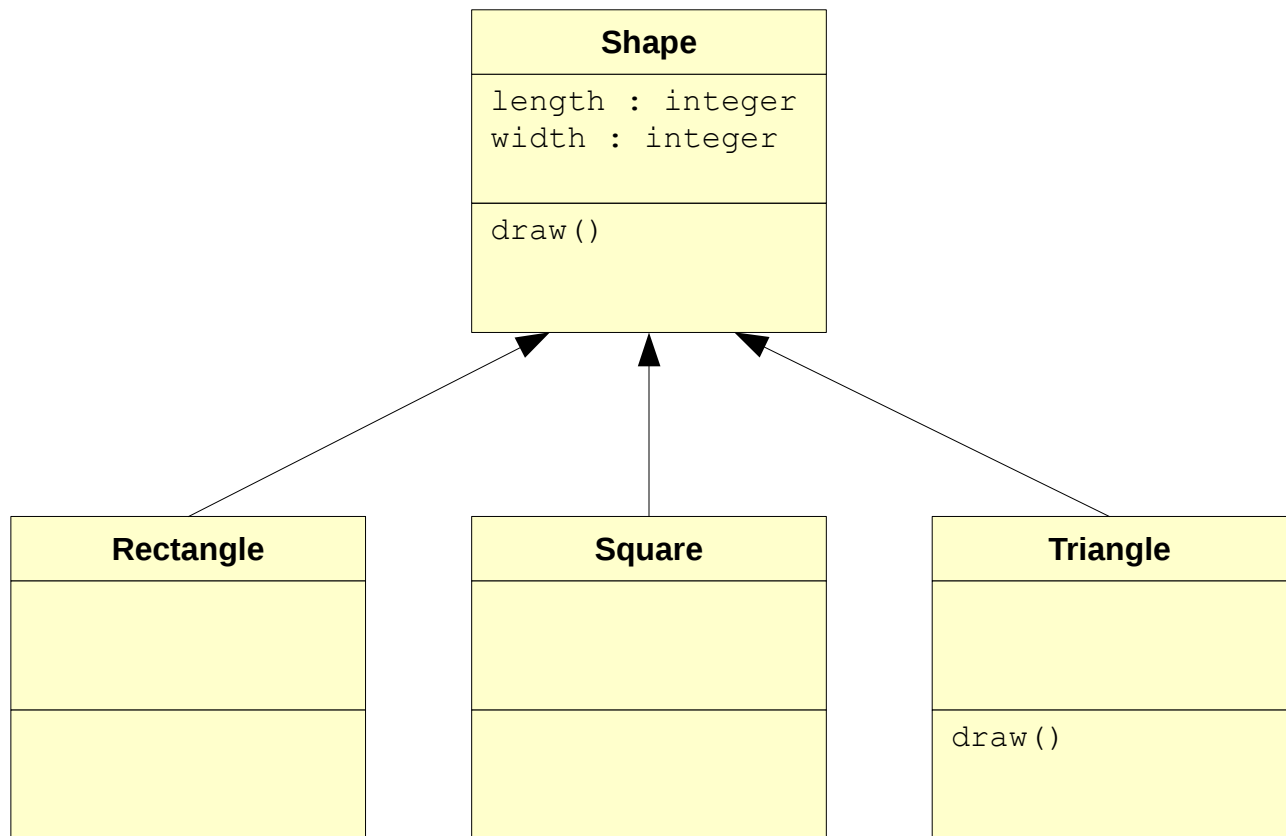
r1 = Rectangle(12, 4)
r1.draw()
print
s1 = Square(6)
s1.draw()

Ln: 1 Col: 0
```

The **Square** and **Rectangle** classes are subclasses of the **Shape** class. Note that they have no instance variables or functions unique to them. That is, they inherit everything from their superclass. The only difference between the two is that the constructor of the **Square** class takes only one argument, while the constructor of the **Rectangle** class takes two arguments. Within their individual implementations, they both call the constructor of the **Shape** class.

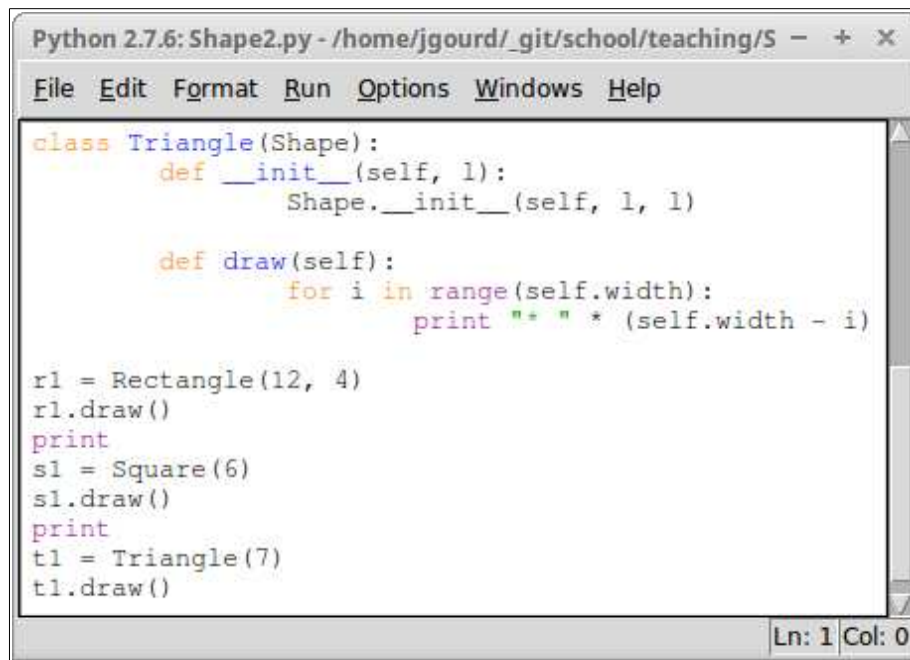
When we want to draw a square or rectangle, we call the `draw` function. Method lookup makes it easy to see that the `draw` function in the **Shape** class will be executed. Why? Because the **Square** and **Rectangle** classes don't have a `draw` function of their own, but their superclass (the **Shape** class) does.

Since one of the benefits of inheritance is to increase code reuse and to ease expansion and application feature enhancement, let's do precisely that by adding the ability to create and draw triangles. To simplify this, let's just consider right-angled isosceles triangles (i.e., the two sides making up the right angle are of equal length). Since triangles are shapes too, it makes sense to make them a subclass of the **Shape** class, yielding the following modified class diagram:



Note the specification of the `draw` function in the triangle class. The shape of a triangle is different from that of a square or a rectangle, and the process of drawing that shape is therefore different. Polymorphism allows us to create another function, also called `draw`, but specifically for triangles. This overwrites the `draw` function specified in the **Shape** class, and effectively specializes the `draw` behavior for a triangle. This version of `draw` would only be executed on an object reference of the type **Triangle**.

Here is the new **Triangle** class, along with an updated main part of the program (note that the rest of the program that defines the other classes remains unchanged):



```

Python 2.7.6: Shape2.py - /home/jgourd/_git/school/teaching/S - + x
File Edit Format Run Options Windows Help

class Triangle(Shape):
    def __init__(self, l):
        Shape.__init__(self, l, l)

    def draw(self):
        for i in range(self.width):
            print "*" * (self.width - i)

r1 = Rectangle(12, 4)
r1.draw()
print
s1 = Square(6)
s1.draw()
print
t1 = Triangle(7)
t1.draw()

Ln: 1 Col: 0

```

The output of this modified program is as follows:

```

* * * * *
* * * * *
* * * * *
* * * * *

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

* * * * *
* * * * *
* * * * *
* * * *
* * *
* *
*

```

Pay close attention to the following statements in the draw function of the **Triangle** class:

```

for i in range(self.width):
    print "*" * (self.width - i)

```

The variable `i` iterates from 0 through the width of the triangle (minus one). Since the triangle is seven units long, then `i` iterates from 0 through 6 (exactly seven times). The first time in the for loop, the variable `i` is equal to 0. Therefore, the number of asterisks displayed is $7 - 0 = 7$. The next time through the loop, `i` is equal to 1, and $7 - 1 = 6$ asterisks are displayed. This continues until the last time through the loop, where `i` is equal to 6 and $7 - 6 = 1$ asterisk is displayed.

Activity 1: Zooland

Now that you have an idea about polymorphism and method lookup, let's look at a hypothetical example in which we'll be more concerned about the placement of the polymorphic methods rather than their actual implementation.

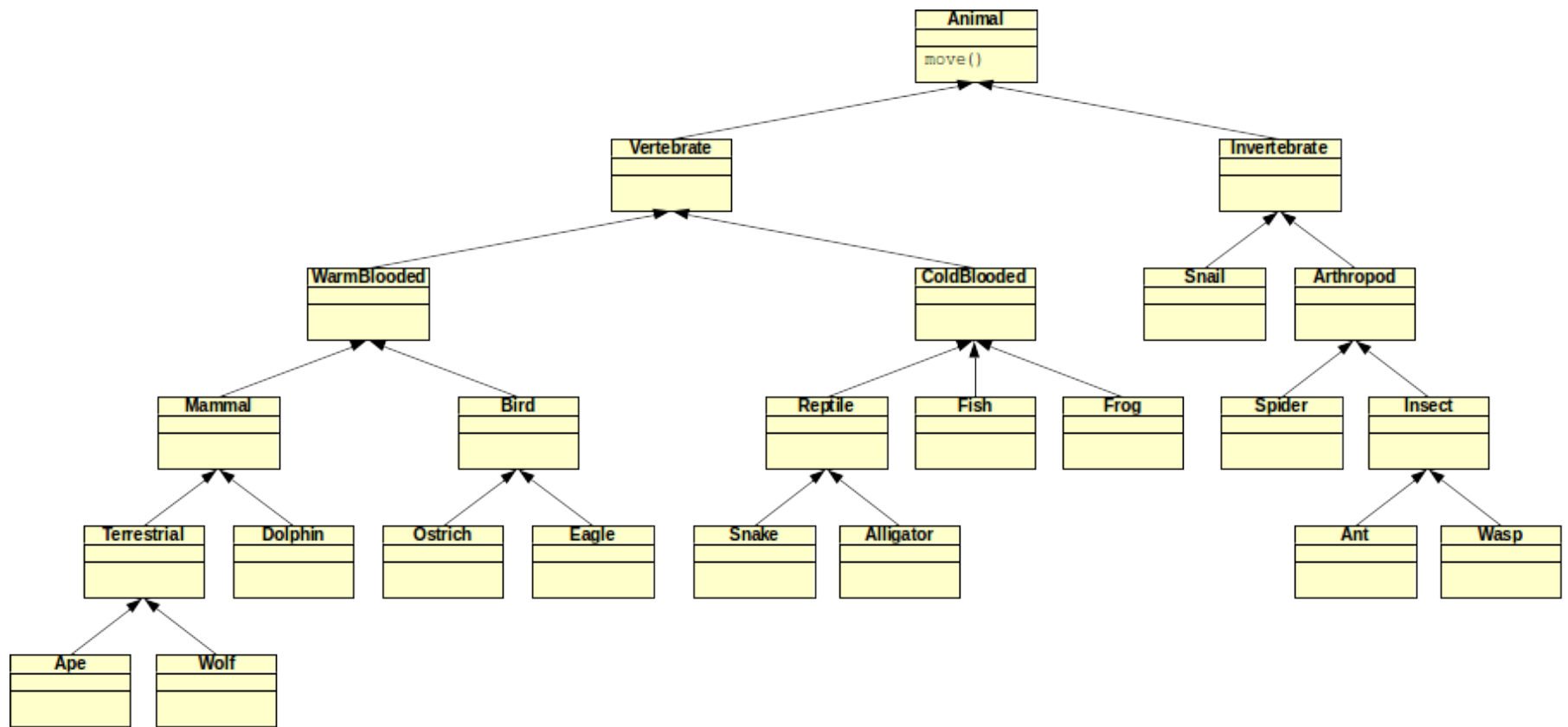
Suppose that you are writing a program to model (i.e., programmatically represent) the types of animals that are in a zoo. Such a situation would easily lend itself to inheritance, since there are multiple animals that are similar in nature (and could therefore inherit similar traits from a superclass). In fact, a possible class diagram for such a program is shown on the next page.

The class diagram shows how a variety of animals are related. All animals move; therefore, a `move` function is defined in the topmost **Animal** class. That particular version of `move` is implemented as: “move in a given direction using four limbs, all of which are in contact with the ground at some point.”

Of course this definition of `move` is not accurate for some of the animals that are in the class diagram. The objective of this activity is to place one of the following alternate versions of `move` in the appropriate classes, such that all animals move in their proper way. Since the motivation behind inheritance is primarily to reduce code duplication, **the goal is to place as few `move` functions in the hierarchy as possible.** Here are the alternate versions of the `move` function :

- (1) Move in a given direction using four limbs, all of which are in contact with the ground at some point (note that this is the version in the animal class);
- (2) Move in a given direction using two limbs, both of which are in contact with the ground at some point;
- (3) Move in a given direction using wings or wing-like body parts;
- (4) Move in a given direction using six or more limbs, all of which are in contact with the ground at some point;
- (5) Move in a given direction using fins or fin-like body parts; and
- (6) Move in a given direction by slithering on the ground.

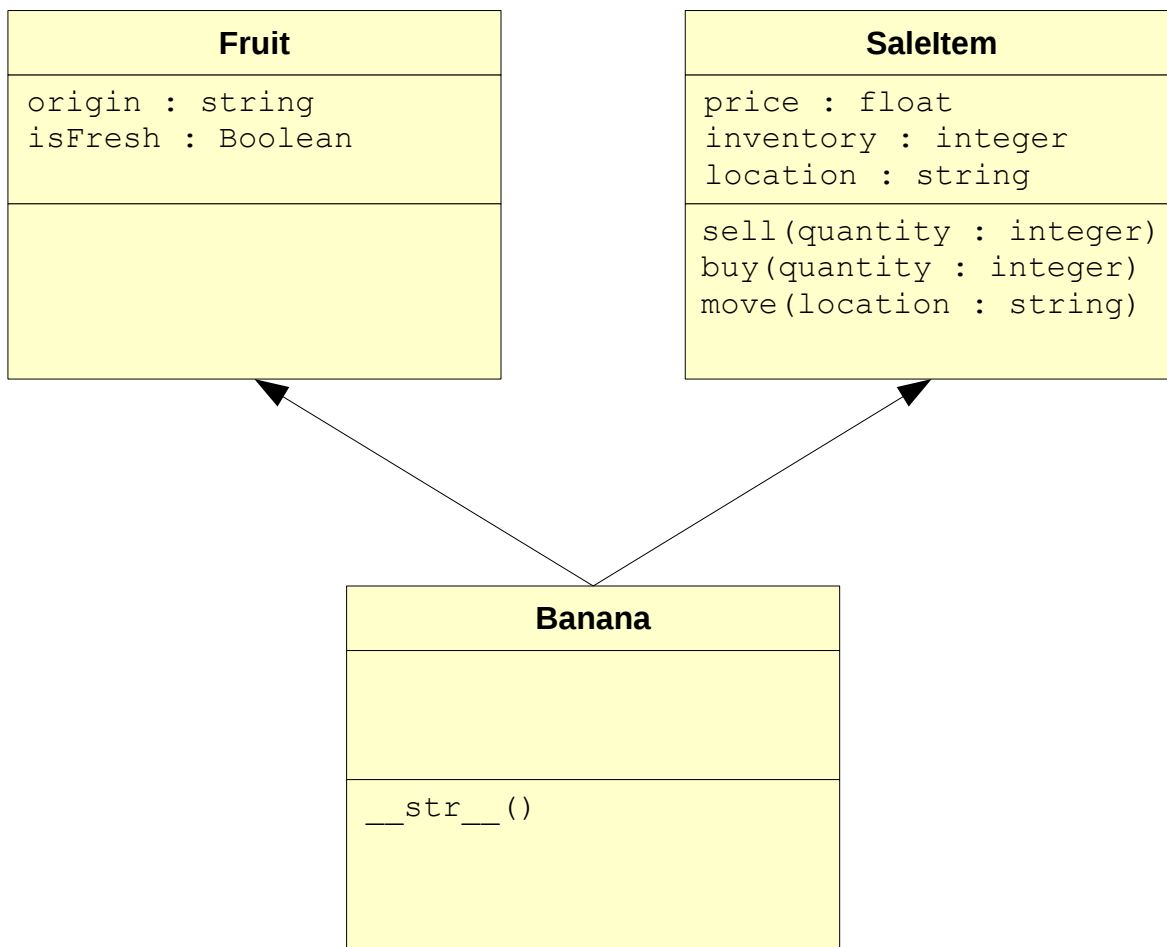
Remember that the higher up a function is in the inheritance hierarchy, the more classes it can be applied to. In addition, it is possible that a better result is obtained by removing or changing the version of `move` currently in the **Animal** class to another version. Note that in cases where an animal could potentially implement more than one of the given versions of `move`, assume that the animal only uses the more dominant version. For example, while an eagle could walk on two limbs, it predominantly flies through the air to move; therefore, use version (3).



Multiple inheritance

In the previous lesson on the object-oriented paradigm, the concept of multiple inheritance (and how it differs from single inheritance) was briefly discussed. In this lesson, we have only looked at cases where a class only inherits traits from a single superclass (i.e., **single inheritance**). Most programming languages only support single inheritance; however, there are cases where it would be advantageous to support inheriting traits from more than one superclass (i.e., **multiple inheritance**).

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



To declare a class as having more than one superclass, we simply put the names of its superclasses (each separated by a comma) in the parentheses following its class name. Here's an example with the banana class:

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

Method lookup in the context of multiple inheritance raises an interesting question: which function is called if more than one superclass has a function with the same name? For example, the **Banana** class does not have the `__str__` function as shown in the class diagram; however, both the **Fruit** and **SaleItem** classes do. So which of the two `__str__` functions is executed supposing some print statement on an object reference of the **Banana** class is executed?

Since the **Banana** class has no `__str__` function, then the normal behavior is to find the matching function in the superclass. However, the banana class has two superclasses. With multiple inheritance, method lookup is carried out in the order in which the superclasses are listed in the class arguments. In the **Banana** class header shown above, for example, the matching function would first be searched for in the **Fruit** class since it is the first superclass listed. If found (which it clearly is), it is executed; if not, then the matching function would be searched for in the **SaleItem** class. If found, it would be executed; if not, an error would occur (since the method would never have been found).

Abstract methods

One of the benefits of the object-oriented paradigm in programming is that it easily allows for more than one programmer to be involved in the implementation of a system. The implementation is often divided according to classes (e.g., one team of programmers works on one class while another team deals with another class). In these cases, it is important to include measures that allow any programmer to know the expectations of each class. One of the measures that helps with this is the idea of abstract methods.

Consider the animal hierarchy that was discussed earlier in this lesson. Suppose that one team was tasked with the design of the **Animal** class, and another with the design of various classes at the bottom of the hierarchy. Generally, it is good programming practice to put the members that all (or many of) the classes will use higher up in the hierarchy. Suppose that the project required a function to represent the way animals communicate (i.e., sound and motion cues). We know that all animals communicate in one way or another; however, there is no general way that applies to the majority of the animals. In this scenario, it is beneficial to use an abstract method.

An **abstract method** in a class is a way of promising that any subclasses of some superclass will provide implementation details for that method. In the communication example above, the **Animal** class has no general way of implementing some sort of communicate function. Since all animals communicate, it is imperative that all subclasses provide an explicit implementation for the communicate function.

A method in a class is made abstract by only including its signature. The only allowable code in the abstract method is an error that can be shown if the subclass has not implemented the abstract method. In Python (as in many other object-oriented programming languages), errors can be *raised* or *thrown* to alert the user of some sort of problem. In Java, for example, errors (actually called exceptions) are thrown. In Python, errors are raised. This will be discussed thoroughly in a future lesson. For now, it is sufficient to note that abstract methods can raise an error known as a **NotImplementedError**. Should a subclass not implement a function defined as abstract in the superclass, and an object reference of the subclass tries to call it, the error will be raised. This terminates the program.

Here is an example of a simple **Animal** class (with the abstract method `communicate`) and a **Bird** subclass. The **Bird** subclass does not implement the abstract method in this example:


```
Python 2.7.6: abstract1.py - /mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Objects/col
File Edit Format Run Options Windows Help

class Animal(object):
    def __init__(self):
        """ Constructs a new Animal """

    def communicate(self):
        """ How an animal communicates """
        raise NotImplementedError("Abstract method communicate not implemented in subclass!")

class Bird(Animal):
    def __init__(self):
        """ Constructs a new Bird """

b = Bird()
b.communicate()

Ln: 1 Col: 0
```

The functions above simply have comments instead of any actual implementation; however, they can still be used to understand the concept. The **Animal** class has a constructor and an abstract function called `communicate`. Making that function abstract is a way of enforcing that all subclasses have an actual implementation of it (instead of, for example, hoping that whoever is working on that class remembers to implement it). Note the only statement in the function:

```
raise NotImplementedError("Abstract method...")
```

The **Bird** class has not implemented the `communicate` function in this example. Therefore, the statement `b.communicate()` causes a **NotImplementedError** to be raised:

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help

Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>

Traceback (most recent call last):
  File "/mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Object
s/code/abstract1.py", line 14, in <module>
    b.communicate()
  File "/mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Object
s/code/abstract1.py", line 7, in communicate
    raise NotImplementedError("Abstract method communicate not implemented in
subclass!")
NotImplementedError: Abstract method communicate not implemented in subclass!
>>> |

Ln: 13 Col: 4
```

But why is the error raised? It all comes down to method lookup. Since the function is not defined in the **Bird** subclass, then it is searched for in its superclass (i.e., the **Animal** class). Of course, it is found there; however, it only contains the statement that raises the error. And so the error is raised!

To show that actually implementing the `communicate` function in the **Bird** class works, here's a modification of the above program:

```
Python 2.7.6: abstract2.py - /mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Objects/cod
File Edit Format Run Options Windows Help

class Animal(object):
    def __init__(self):
        """ Constructs a new Animal """

    def communicate(self):
        """ How an animal communicates """
        raise NotImplementedError("Abstract method communicate not implemented in subclass!")

class Bird(Animal):
    def __init__(self):
        """ Constructs a new Bird """

    def communicate(self):
        """ How a bird communicates """
        print "A Bird communicates!"

b = Bird()
b.communicate()

Ln: 1 Col: 0
```

The output of the program is the single line:
A Bird communicates!

Another way of implementing an abstract method is by making use of the **Abstract Base Class** (abc) library that is packaged with Python. When imported, a method can be marked as abstract in the same way that one can be marked as an accessor or mutator. The difference is the decorator used to mark the function: `@abc.abstractmethod`. This method ensures that a **TypeError** will be raised immediately upon the instantiation of a subclass if the abstract method is not implemented in the subclass. Note that the following statement must be placed at the top of the class (i.e., after the class signature) in order for this method to work:

```
__metaclass__ = abc.ABCMeta
```

Here's an example of a modified **Animal** class that implements this way of including abstract methods:

```
Python 2.7.6: abstract3.py - /mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Objects/cod
File Edit Format Run Options Windows Help

import abc

class Animal(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self):
        """ Constructs a new Animal """

    @abc.abstractmethod
    def communicate(self):
        """ How an animal communicates """

class Bird(Animal):
    def __init__(self):
        """ Constructs a new Bird """

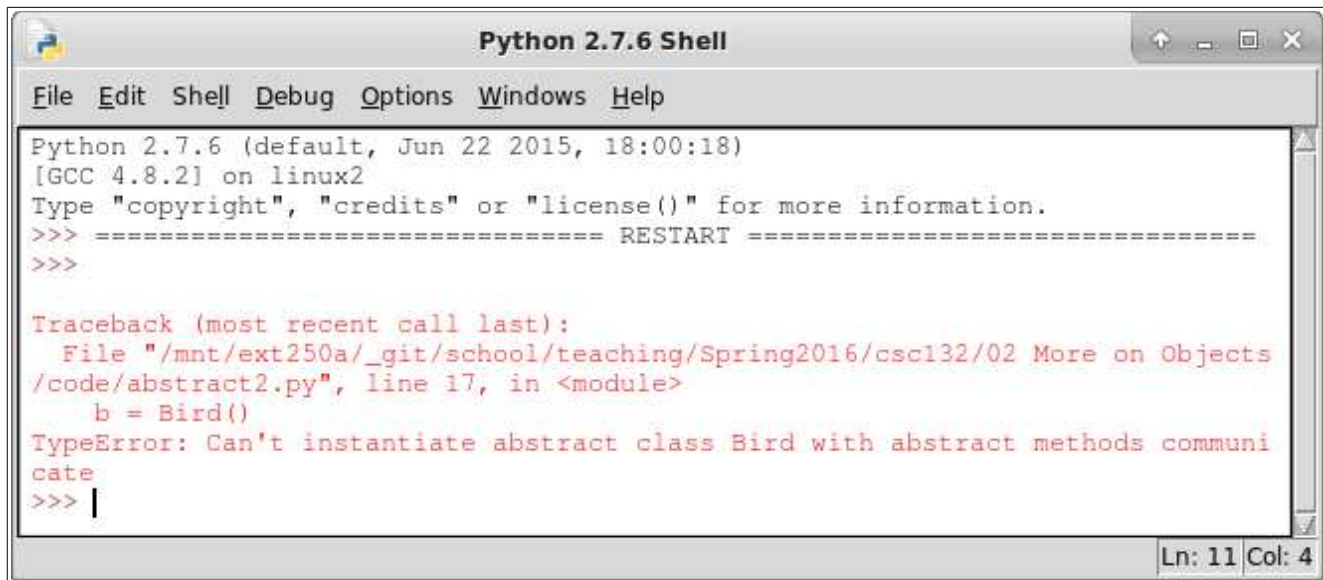
b = Bird()
b.communicate()

Ln: 1 Col: 0
```

Note the differences. First, the **abc** library is imported. Second, the `communicate` method is marked as abstract via `@abc.abstractmethod`. Third, the class has the following statement placed immediately after its signature:

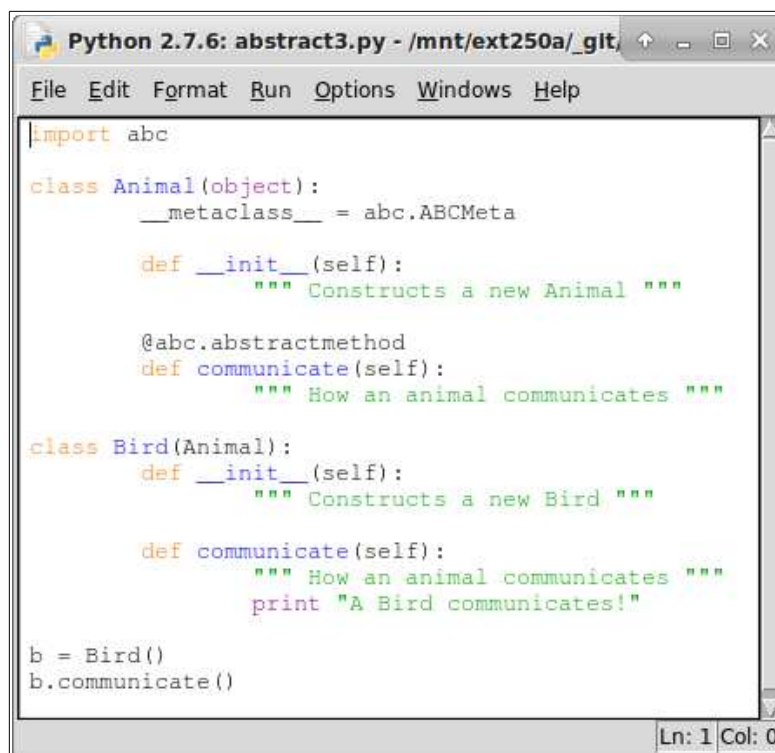
```
__metaclass__ = abc.ABCMeta
```

When the variable `b` is instantiated as a new object of the **Bird** class, the following error occurs:



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "/mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Objects
/code/abstract2.py", line 17, in <module>
    b = Bird()
TypeError: Can't instantiate abstract class Bird with abstract methods communi
cate
>>> |
```

To show that actually implementing the abstract method in the **Bird** subclass works with this way of including abstract methods, here is working source code:



```
Python 2.7.6: abstract3.py - /mnt/ext250a/_git
File Edit Format Run Options Windows Help
import abc

class Animal(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self):
        """ Constructs a new Animal """

    @abc.abstractmethod
    def communicate(self):
        """ How an animal communicates """

class Bird(Animal):
    def __init__(self):
        """ Constructs a new Bird """

    def communicate(self):
        """ How an animal communicates """
        print "A Bird communicates!"

b = Bird()
b.communicate()

Ln: 1 Col: 0
```

The output of the program is the single line:

A Bird communicates!

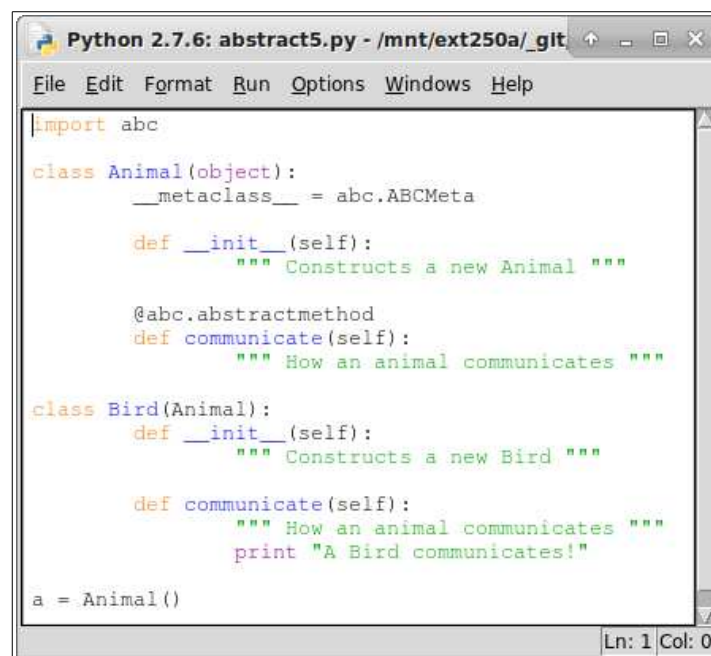
In standard practice, the first method is usually preferred. The main reason for this is that the second is not particularly *Pythonic*. Python prides itself in being dynamic and obvious. That is, it is quite readable that the `communicate` method defined in the **Animal** class must be implemented in the **Bird** subclass because it raises an error if it is called. On the other hand, the first method requires the unimplemented function to actually be called by an object reference of the first class before an error is raised. The second method immediately raises an error when an object of the subclass that doesn't implement the abstract method is instantiated. So in the end, take your pick.

Abstract classes

Recall the Zooland activity discussed earlier. Although the **Animal** class exists and can be instantiated, what kind of animal would such a instance imply? That is, what kind of animal would it be? In fact, it could be any animal. The intended behavior is most likely to only allow the creation of instances of the various animals at the bottom of the hierarchy. For example, instantiating an **Ostrich**, **Dolphin**, and **Wolf** should be allowed. However, instantiating a **Bird**, **Mammal**, or **Terrestrial**(Mammal) should not since they aren't specific enough and really describe groups or types of animals.

Instantiating any class in the hierarchy can create confusion because it could potentially allow instances of objects that were not planned in the design. We can restrict the instantiation of the **Animal** class, for example, by making it abstract. That way, no one can just instantiate an **Animal** object. **Abstract classes** are classes designed solely to be used as superclasses and never to be instantiated. Ensuring that the subclasses of abstract classes are not abstract means that they can be instantiated.

In Python, the only way to make a class abstract is to use the Abstract Base Class method discussed above. To prevent the class from actually being instantiated, at least one of its methods must be made abstract. The following **Animal** class, for example, cannot be instantiated because its `communicate` method is abstract:



```
Python 2.7.6: abstract5.py - /mnt/ext250a/_git
File Edit Format Run Options Windows Help

import abc

class Animal(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self):
        """ Constructs a new Animal """

    @abc.abstractmethod
    def communicate(self):
        """ How an animal communicates """

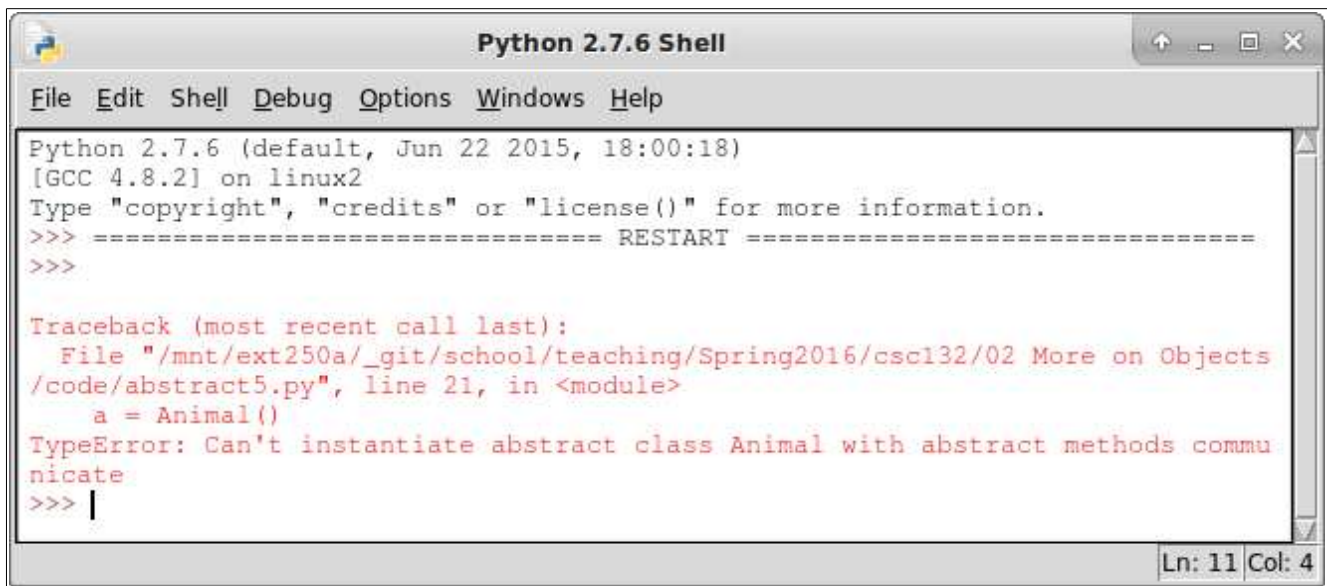
class Bird(Animal):
    def __init__(self):
        """ Constructs a new Bird """

    def communicate(self):
        """ How an animal communicates """
        print "A Bird communicates!"

a = Animal()

Ln: 1 Col: 0
```

Note that the program above is the same as before, except that the main part of the program has been changed to attempt to declare an instance of the **Animal** class. Here's the output:

A screenshot of a Python 2.7.6 Shell window. The window has a title bar "Python 2.7.6 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following output:

```
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>

Traceback (most recent call last):
  File "/mnt/ext250a/_git/school/teaching/Spring2016/csc132/02 More on Objects
/code/abstract5.py", line 21, in <module>
    a = Animal()
TypeError: Can't instantiate abstract class Animal with abstract methods commu
nicate
>>> |
```

The status bar at the bottom right shows "Ln: 11 Col: 4".

Make sure that you understand that using the **abc** library to implement abstract methods means that the entire class is also abstract. However, it is the only way to actually make an entire class abstract. So which method should you choose? If you want to make entire classes abstract, then you must use the second method (i.e., using the **abc** library). If you just want to make one or more functions in a class abstract but not the entire class, then use the first method.

Did you know?

As discussed above, abstract classes cannot be instantiated, and are instead only used as superclasses to define state and behavior that can be inherited by subclasses. **Concrete classes** are those classes that can be instantiated.

Coupling and cohesion

So far in this lesson, the concepts that have been discussed all help in designing code. In particular, you may have noticed that they help reason about and design large projects involving many programmers. The kinds of projects that you will likely work on in an educational setting are usually small in comparison to the kinds of projects that professionals work on in industry. These large projects require careful consideration when designing them, programming them, and later testing them, so that they are manageable and ultimately produce a solution that solves some original problem.

The rest of this lesson will focus on a few more topics that help address the design of applications. The first looks at the connections between separate units of a program. A **unit** is any reasonably self-contained component of a program. For example, a class can be considered a unit. But at another level of abstraction, so can a function. In fact, the function can be considered as zooming in on a class. Zooming out of a class may reveal several classes, all contained within some library. The library can also be considered a unit, just at another level of abstraction.

Coupling refers to links between the separate units of a program. If two classes depend closely on one another, we say that they are **tightly coupled**. Similarly, if two methods depend closely on one another, they are tightly coupled. Two units that do not depend closely on one another are **loosely coupled**. When designing applications, we aim for loosely coupled units, or generally just loose coupling. Why? Because it makes it possible to understand one unit without having to understand others. In the end, it just makes things simpler. For example, we can change one class with little to no effect on other classes. This increases maintainability. We try to avoid tight coupling, because changes to a unit that is tightly coupled with one or more other units can cascade and result in a chain of additional changes to the other units.

Cohesion refers to the number and diversity of tasks that a single unit is responsible for. If a unit is responsible for a single task, we say that it has **high cohesion**. If a unit is responsible for many tasks, we say that it has **low cohesion**. When designing applications, we aim for high cohesion. The more simple and lean a unit is, the easier it is to understand what it does. It's just that much easier to reason about and maintain.

Generally, applications that are highly cohesive often make it easy to reuse units in order to reduce code duplication. Low cohesion implies that units perform multiple tasks and have no clear identity. So what does this mean specifically in terms of application design? At the class level: classes should represent one single, well defined entity; and at the function level: a function should be responsible for one well defined task.

Designing applications with loose coupling and high cohesion helps with localizing change. When change is needed, as few units as possible should be affected. In fact, we can use the concepts of coupling and cohesion to answer the questions, “How long should a class be?” and “How long should a function be?” Simply put, a function is too long if it does more than one logical task, and a class is too complex if it represents more than one logical entity. Of course, these are guidelines; the real world is often a bit more complicated.

In previous lessons that dealt with computer architecture, many fundamental building blocks of computers were covered; for example:

- The layers of a computer system (e.g., operating system, applications, and so on);
- Fundamentals of digital logic (circuits and switches);
- Logic gates (*and*, *or*, and *not*);
- Boolean algebra;
- Combinational circuits (*xor*, comparators, and adders);
- The binary and hexadecimal number systems;
- Converting between number systems;
- Binary arithmetic (addition and multiplication); and
- Half and full adders, and chaining full adders together to add larger binary numbers.

In this lesson, we will first discuss how numbers and characters are represented in computers. So far, all of the numbers that we have looked at have been *unsigned* (i.e., assumed to be positive). For example, the number 1001001_2 is equivalent to 73_{10} . However, in order to be able to handle negative numbers, we need a way to represent the sign of a number. Subsequently, we will cover several more combinational circuits that prove quite useful in the design of computing systems. In addition, several important sequential circuits that serve as the building blocks for computer memory will be discussed. Finally, we will show how a Central Processing Unit (CPU) is built from the primitive components discussed in the lessons on computer architecture.

Signed numbers

An early attempt to handle signed numbers was to add a special sign *bit* to the left of each number. A zero in the sign bit was used for positive numbers, and a one in the sign bit was used for negative numbers. The representations of +5 and -5 in **signed magnitude notation** are shown below. These representations assume that three bits are reserved for the magnitude of a number and one bit for its sign:

Signed magnitude binary				Decimal
Sign	Magnitude			
0	1	0	1	+5
1	1	0	1	-5

Signed magnitude notation works, provided that the sign bit is treated separately from the magnitude of the number and guides how arithmetic operations are performed. However, arithmetic in this notation is difficult and results in some anomalies. For example, there are two representations for zero (+0 is 0000 and -0 is 1000). To illustrate the difficulties in arithmetic, let's look at the sum of +5 and -5 (which should be 0):

	Binary				Decimal
Carry	1	1	0	1	
1st number		0	1	0	+5
2nd number		1	1	0	-5
Sum	1	0	0	1	2?

Note that the sum (10010) does not equal 0 (0000 or 1000) using signed magnitude notation (even if we ignore the leftmost bit). Arithmetic becomes easier, however, if we write a negative number as the complement of the corresponding positive number. The **complement** of a binary number is formed by writing a 0 wherever there is a 1 in the original number, and a 1 wherever there is a 0 in the original number. As with signed magnitude notation, the leftmost bit still indicates the sign of the number: 0 for positive numbers, and 1 for negative numbers. Complementing a binary number in this way to represent signed values is sometimes referred to as **one's complement notation**. The one's complement representation of +5 and -5 are shown below:

One's complement binary				Decimal
Sign	Magnitude			
0	1	0	1	+5
1	0	1	0	-5

Although the arithmetic for one's complement numbers is much easier than for signed magnitude numbers, the anomaly of two representations for zero (0000 for +0 and 1111 for -0) still exists. Let's see how +5 and -5 can be added using one's complement notation:

	Binary				Decimal
Carry	0	0	0		
1st number	0	1	0	1	+5
2nd number	1	0	1	0	-5
Sum	1	1	1	1	0

Indeed, the result (1111) is 0 (well, one variation of it: -0). **Two's complement notation** is a variation of one's complement notation that only includes a single representations for 0. To change the sign of a two's complement binary number, we perform the following three steps:

- (1) Write down the binary representation of the original number;
- (2) Complement all of the bits (i.e., replace 1's with 0's and 0's with 1's); and
- (3) Add one to the result.

Here are +5 and -5 written in two's complement notation:

Two's complement binary				Decimal
0	1	0	1	+5
1	0	1	1	-5

Let's see how +5 and -5 can be added using two's complement notation:

	Binary				Decimal
Carry	1	1	1	1	
1st number		0	1	0	+5
2nd number		1	0	1	-5
Sum	<i>1</i>	0	0	0	0

Notice that while we get a correct answer (0000), this problem also generated a carry bit (the italicized leftmost 1). We will discuss this anomaly later. While the idea of two's complement notation may seem a little strange at first, it has a unique representation for zero (0000, given four bits) and straightforward arithmetic operations. Notice that, for positive numbers, all three representations (signed magnitude, one's complement, and two's complement) have the same pattern as unsigned numbers.

The following table presents the sixteen signed numbers that can be represented using two's complement notation and four bits of storage. Notice that the numbers range from negative eight to positive seven. In two's complement notation, the range of numbers that can be represented given a fixed number of bits will always include one more negative value than there are positive values. This is because the representation of zero includes a 0 in the leftmost bit position, thus taking up one of the “positive” slots.

Two's complement binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

For n bits of storage the range of numbers that can be represented in two's complement notation extends from -2^{n-1} to $+2^{n-1}-1$. In the table above, four bits of storage were used; therefore, the range of numbers was from $-2^{4-1} = -2^3 = -8$ to $2^{4-1}-1 = 2^3-1 = 7$. Most modern computers use 32 bits to represent integers. Thus, they are capable of representing values in the range $-2^{31} = -2,147,483,648$ to $2^{31}-1 = 2,147,483,647$.

What do you suppose happens if 1 is added to 7? In other words, how does the computer handle $0111 + 0001$? Performing binary addition, the result is 1000. In the table, this is -8! Does this mean that $7 + 1 = -8$? If the computer stored integers using only four bits, then indeed this would be the case! Counting essentially wraps around the table. That is, $7 + 1 = -8$ and $-8 - 1 = 7$. The four bits of storage only allow numbers in this range.

In order to understand why two's complement is the preferred method for representing signed numbers at the machine level, we will look at a number of addition problems involving both positive and negative numbers. As we will see, what makes two's complement so great is that the sign of a number can essentially be ignored when performing addition (since positive and negative numbers are treated in an identical manner). An added bonus is that we get the subtraction operations for “free” once we have addition: a problem such as $X - Y$ can be recast as the following two step process:

- (1) Swap the sign of Y ; and
- (2) Add X and Y .

Let's begin by examining the summation of numbers with opposite signs. First, the sum of +2 and -3 (which should sum to -1). Assuming four bits of storage, the two's complement binary representation of +2 is 0010. The two's complement representation of -3 is 1101. Why? To obtain -3, we begin with +3 (0011), complement the bits (1100), and add one (1101). Here's the addition of +2 and -3:

	Binary				Decimal
Carry	0	0	0		
1st number	0	0	1	0	+2
2nd number	1	1	0	1	-3
Sum	1	1	1	1	-1

The 1 in the leftmost column of the result tells us that the number is negative. Its magnitude can be determined by complementing each bit (0000) and adding one (0001). Thus, the addition of +2 and -3 is 1111 (or -1).

Next, let's look at the addition of -2 and +3. The two's complement binary representation of -2 is 1110 (start with +2 which is 0010, complement the bits to obtain 1101, and add one to obtain 1110). The representation of +3 is 0011. Adding these two values together gives +1 or 0001 in binary two's complement, as is illustrated below:

	Binary				Decimal
Carry	1	1	1	0	
1st number		1	1	1	-2
2nd number		0	0	1	+3
Sum	<i>1</i>	0	0	0	+1

Note that this particular addition of two four-bit numbers results in a carry to a fifth bit. We saw this before (when adding +5 and -5). This carry is ignored. In order to reinforce the fact that this bit is discarded, it is shown in italics.

Let's now turn our attention to addition problems involving numbers of the same sign. Here is an illustration of the addition of two positive numbers: +2 (0010) and +3 (0011):

	Binary				Decimal
Carry	0	1	0		
1st number	0	0	1	0	+2
2nd number	0	0	1	1	+3
Sum	0	1	0	1	+5

To show that the system handles the addition of negative numbers properly, consider adding -2 (0010 → 1101 → 1110) and -3 (0011 → 1100 → 1101). The result should be -5 (0101 → 1010 → 1011):

	Binary				Decimal
Carry	1	1	0	0	
1st number		1	1	1	-2
2nd number		1	1	0	-3
Sum	<i>1</i>	1	0	1	-5

The result is indeed 1011. The 1 in the leftmost bit indicates that the result is a negative number. We can determine its magnitude by implementing the two's complement operations on it: complement the bits 1011 to obtain 0100, and add one to obtain 0101.

Notice that while we get a correct answer, this problem also generated a carry bit that is discarded. One problem that can arise when representing numeric values via a fixed number of bits is the problem of overflow. **Overflow** occurs when the value that is to be stored is outside the range of permissible values (i.e., the value is too large to fit in the available space). The only way around overflow is to add more bits to the representation, thus increasing the range of permissible values. The best we can do in place of this is to detect when overflow occurs.

Two's complement notation makes it easy to spot when overflow occurs: when two numbers of the same sign are added together and the result has the opposite sign. Here are two examples; first, the sum of +4 and +5 (which are both positive numbers):

	Binary				Decimal
Carry	1	0	0		
1st number	0	1	0	0	+4
2nd number	0	1	0	1	+5
Sum	1	0	0	1	-7?

Clearly, the result is not -7. Note that the result (-7) has a different sign than the two numbers (+4 and +5). Here's another example:

	Binary				Decimal
Carry	1	0	0	0	
1st number		1	1	0	-4
2nd number		1	0	1	-5
Sum	1	0	1	1	+7?

Overflow in two's complement can be spotted when the sign bits of both numbers being added are the same, yet the sign bit of the answer is different. Note that overflow can only occur when adding numbers of like sign. It can never occur when numbers of opposite signs are added.

When an overflow is detected, the answer is incorrect, as shown above by the question marks, and must be discarded. There is no way to get the correct answer if the number of bits available does not allow us to express that answer. There is no way to express +9 or -9 (the correct results to the examples above) using a four-bit two's complement number, since both are outside the range of permissible values.

Characters

We now turn our attention to representing characters at the machine level. One of the most popular methods of representing character data in a computer is to use the ASCII character set. The basic idea behind **ASCII** (American Standard Code for Information Interchange), and in fact all other character sets, is to associate particular bit patterns with individual characters.

There are 128 symbols in the standard ASCII character set. These are numbered from 0 to 127. ASCII characters are grouped together in the following way:

Character group	Range	
	Decimal	Hexadecimal
Control characters	0 – 31	0 – 1F
Punctuation	32 – 47	20 – 2F
Digits	48 – 57	30 – 39

Character group	Range	
	Decimal	Hexadecimal
More punctuation	58 – 64	3A – 40
Uppercase letters	65 – 90	41 – 5A
More punctuation	91 – 96	5B – 60
Lowercase letters	97 – 122	61 – 7A
More punctuation	123 – 126	7B – 7E
One final control character	127	7F

Control characters refer to *nonprinting* characters (e.g., return/enter, linefeed, delete, and so on). Punctuation refers to any printable character that is not a digit or letter (e.g., [, {, |, \, and so on).

The following table contains the hexadecimal representations of all of the printable ASCII characters. These range from the space character (with an ASCII value of $20_{16} = 32_{10}$) to the tilde (with an ASCII value of $7E_{16} = 126_{10}$). The ASCII value for a character may be found in the table below by locating the row and column in which the character appears. The row specifies the high-order (leftmost) hexadecimal digit, and the column specifies the low-order (rightmost) hexadecimal digit. For example, the ASCII value for the uppercase letter H is 48_{16} , since it appears in row 4 and column 8. To determine the decimal version of the ASCII value, you must perform the conversion from hexadecimal to decimal (e.g., $48_{16} = 72_{10}$).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	[space]	!	“	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

While any assignment of bit patterns to characters would accomplish the primary task of enabling computers to store character data, a lot of thought went into the design of ASCII. For example, uppercase and lowercase characters differ in ASCII by exactly one bit (e.g., $A = 41_{16} = 01000001_2$, and $a = 61_{16} = 01100001_2$). Therefore, any lowercase character can be changed to upper case by simply setting bit six to 0. Conversion from uppercase to lowercase is just as easy: set bit six to 1!

The concept of mapping characters to bit patterns is logical; however, characters typed on a keyboard don't show up on the screen as ASCII values. This is because ASCII values are hidden from the computer user. When you type “patootie” on your keyboard, you first hit a key marked “p”. This causes the keyboard circuitry to generate the ASCII code 70_{16} . Next, you strike the key marked “a”, causing the keyboard circuitry to generate the ASCII code 61_{16} . This process occurs for the entire word. The display circuitry built into your computer's video card then displays the characters of the word as it

receives the ASCII values for each character. Since the internal circuitry of both the keyboard and video card handle the translation and processing of ASCII values, you never see the ASCII values.

It is interesting to note what happens when the shift key is held down as, for example, the word “patootie” is typed. What the shift key actually does (at least for the keys marked with letters of the alphabet) is to set bit six to 0. Thus, the combination of *shift* and *p* (which is $70_{16} = 01110000_2$) produces $50_{16} = 01010000_2$ (which is the ASCII value for *P*).

Floating point numbers

Most modern personal computers are based on a bus size of thirty-two bits. Using two’s complement notation, they can conveniently manipulate numbers in the range of approximately -2 billion to +2 billion. Many times, we humans need our computers to work with numbers far outside of this relatively narrow range. For example, scientific applications often need to represent very small or very large quantities, such as Avogadro’s number: $6.02 * 10^{23}$ molecules/mole. Even a program designed to make projections about the national debt of the United States, which is measured in trillions of dollars, will be forced to work with numbers outside of the range provided by the 32-bit two’s complement representation. In addition to the problem of being limited to a relatively narrow range of values, the two’s complement binary notation we studied cannot represent fractions. This makes it impossible to express concepts like one-half and 75%.

Floating point notation, which is closely related to exponential notation, addresses both of these problems. As you will recall from math or science classes, exponential notation is often used to express very small or very large numbers. For example, the speed of light can be expressed as $3.0 * 10^8$ meters/second.

The two components necessary to express a number in exponential notation are called the **mantissa** (3.0 in the above number representing the speed of light) and the **exponent** (8 in the above number representing the speed of light). These values are expressed in base ten, so the exponent is expressed in terms of base ten as well. Essentially, the exponent tells us the number of positions that the decimal point should be moved to the right or left. Positive exponents cause the decimal point to move to the right; negative exponents cause the decimal point to move to the left. Hence, the representation for the speed of light translates to a 3 followed by eight 0s (i.e., three hundred million meters per second). Fractions can also be expressed in this system; for example, $1/1000 = 1.0 * 10^{-3} = 0.001$.

Floating point notation is the machine level equivalent of exponential notation. Floating point numbers include signed representations of both the mantissa and the exponent. Since the representations of both will be in base two, it will be considered the base of the exponent.

As an example, here is the number 56.0 expressed using one possible 32-bit floating point representation:

+ -	Exponent								Mantissa																											
	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1			
	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1				

The leading bit (bit 32) of this representation signifies the sign of the number (0 for positive, 1 for negative). Specifically, it represents the sign of the mantissa. The next eight bits (bits 31 to 24) represent the exponent in two's complement notation (remember the exponent can be positive or negative). The final 23 bits are the mantissa expressed as an unsigned number. The number represented in this example (in binary form) is $111 * 2^{11}$ which in decimal form becomes $7 * 2^3 = 7 * 8 = 56.0$.

Another way of thinking about this example is to view the exponent as specifying the number of 0s that are to follow the bit pattern supplied by the mantissa. In this case, we have 111 followed by three 0s, or $111000_2 = 56$.

Here is an example of a number that is outside the range of values that can be stored using a 32-bit two's complement notation. The number (in binary form) is $11 * 2^{100000}$, which in decimal form is $3 * 2^{32} = 12,884,901,888$:

+ -	Exponent								Mantissa																											
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1					

Another way of thinking about the number represented by this bit pattern is 11 followed by thirty-two 0s. Now, consider the representation of -0.75 as a floating point number:

+ -	Exponent								Mantissa																						
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

The 1 in the leftmost bit signifies that the number is negative. The exponent, 11111110_2 is also negative due the 1 in its leftmost bit (remember that the exponent is represented in two's complement notation). The magnitude of the exponent is 2 (complement the bits to get 00000001_2 , and add one to get $00000010_2 = 2_{10}$). The magnitude of the mantissa is $11_2 = 3_{10}$. Therefore, the value is:

$$-3 * 2^{-2} = -3 * \frac{1}{2^2} = -3 * \frac{1}{4} = -3 * 0.25 = -0.75$$

Another way of thinking about the number in this example is to view it (in binary form) as -0.11. The columns to the right of the decimal point are the 1/2's column, the 1/4's column, the 1/8's column, and so on. The number -0.11 would then be interpreted as having a 0 in the one's place, a 1 in the 1/2's place, and a 1 in the 1/4's place, producing -3/4 (or -0.75). Using this interpretation, the mantissa always has an implied decimal point immediately to its right. In the example above, this is viewed as 11. (note the decimal point to the right). The exponent then specifies the number of places that the decimal point should be moved (to the right for positive exponents, or to the left for negative exponents). In the example above, the exponent moves the decimal point two places to the left giving 0.11. The binary number is then interpreted in the following way:

8	4	2	1		
0	0	0	0	1/2	1/4
				1	1

To work backwards (i.e., to convert -0.75 to binary), we must first break the number down into two parts, each separated by the decimal point. The part of the number that is to the *left* of the decimal point is easily converted using the remainder method for unsigned numbers shown in a previous lesson. Recall, that we repeatedly divide the decimal number by two and build its binary representation by using the remainders generated, from right-to-left. In this case, however, it's simple in that $0_{10} = 0_2$.

The part of the number that is to the *right* of the decimal point is effectively done in reverse when compared to the remainder method. That is, it is repeatedly *multiplied* by two. The binary representation is built by using the bits generated to the left of the decimal point, from left-to-right. Specifically, we multiply the decimal number by two and separately record both parts of the result to the left and right of the decimal point. We then repeat this process with the part to the right of the decimal point, while keeping track of the parts to the left. This is repeated until the part to the right of the decimal point is 0. The binary equivalent of the original number (remember that it's the part to the right of the decimal point in the original number) is subsequently given by listing the parts to the left of the decimal point in the order of their derivation (i.e., from the first identified to the last).

This may be a bit confusing, so let's try this method on the decimal number -0.75 . As shown above, the 0 to the left of the decimal point is easily converted to binary: 0. We'll use the following table to convert the part to the right of the decimal point, $.75$:

Value	Times	Two	Equals	Product	Left part	Right part
.75	*	2	=	1.5	1	.5
.5	*	2	=	1.0	1	.0

Since the part to the right of the decimal point is 0, the process is finished. Combining the “left part” results produces 11 as the part to the right of the decimal point in the final answer. To complete the conversion, we concatenate the parts to the left and right of the decimal point. Therefore, the binary representation of -0.75_{10} is -0.11_2 . From this point, we could generate the table shown earlier that breaks -0.75 down into a sign bit, an exponent, and a mantissa. We know that the sign bit is 1:

+ -	Exponent							Mantissa																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
1																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						

The mantissa is obtained by moving the decimal point all the way to the right: $.11 \rightarrow 1.1 \rightarrow 11$. The mantissa is therefore 11. We also note that it was moved two decimal places:

+	Exponent								Mantissa																							
-																																
1									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1			

Lastly, we know that the exponent is -2 (since we'll need to move the decimal point 2 places to the left in the mantissa). We know how to convert -2 to binary by starting with the binary representation of 2, complementing the bits, and finally adding one: $00000010 \rightarrow 11111101 \rightarrow 11111110_2$:

+	Exponent									Mantissa																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																										
-																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																				</

Let's try another example and convert +5.5625 to binary. First, the 5 to the left of the decimal point: $5_{10} = 101_2$. Next, we'll convert the part to the right of the decimal point using the method described earlier:

Value	Times	Two	Equals	Product	Left part	Right part
.5625	*	2	=	1.125	1	.125
.125	*	2	=	0.25	0	.25
.25	*	2	=	0.5	0	.5
.5	*	2	=	1.0	1	.0

Therefore, $5.5625_{10} = 101.1001_2$. Similarly, we can fill the table below, specifying the sign bit, exponent, and mantissa. First, the number is positive; therefore, the sign bit is 0:

+	Exponent								Mantissa																							
-																																
0																																

Next, the mantissa is 1011001 (we shifted the decimal point all the way to the right):

+	Exponent								Mantissa																								
-																																	
0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1

Finally, the exponent is -4 (since the decimal point needs to be moved four places to the left). We convert -4 to binary using two's complement notation: $00000100 \rightarrow 11111011 \rightarrow 11111100_2$:

+ -	Exponent								Mantissa																					
0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1

Working in reverse, we can actually recreate the decimal representation of the number. We know that the number is positive because the sign bit is 0. We can convert the exponent using two's complement notation: $11111100 \rightarrow 00000011 \rightarrow 00000100_2 = -4$. And we can convert the mantissa: $1011001_2 = 89_{10}$.

We then combine the components into exponential notation (using two as the base for the exponent):

$$89 * 2^{-4} = 89 * \frac{1}{2^4} = 89 * \frac{1}{16} = 89 * 0.0625 = 5.5625$$

Let's try another example, converting -16.125 to binary. First, the left part: $16_{10} = 10000_2$. Next, the right part:

Value	Times	Two	Equals	Product	Left part	Right part
.125	*	2	=	0.25	0	.25
.25	*	2	=	0.5	0	.5
.5	*	2	=	1.0	1	.0

Therefore, $-16.125_{10} = 10000.001_2$. As before, we can fill the floating point table. First, the number is negative; therefore, the sign bit is 1:

+	Exponent								Mantissa															
-																								
1																								

Next, the mantissa is 10000001 (we shifted the decimal point all the way to the right):

+	Exponent								Mantissa																			
-																												
1									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Finally, the exponent is -3 (since the decimal point needs to be moved three places to the left). We convert -3 to binary using two's complement notation: $00000011 \rightarrow 11111100 \rightarrow 11111101_2$:

+	Exponent								Mantissa																							
-																																
1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	

Again, we can recreate the decimal representation of the number. We know that the number is negative because the sign bit is 1. We can convert the exponent: $11111101 \rightarrow 00000010 \rightarrow 00000011_2 = -3$. And we can convert the mantissa: $10000001_2 = 129_{10}$. We then combine the components:

$$-129 * 2^{-3} = -129 * \frac{1}{2^3} = -129 * \frac{1}{8} = -129 * 0.125 = -16.125$$

Let's try one last (perhaps more complicated) example, converting -2337.7109375 to binary. First, the left part: $-2337_{10} = 100100100001_2$. Now, the right part:

Value	Times	Two	Equals	Product	Left part	Right part
.7109375	*	2	=	1.421875	1	.421875
.421875	*	2	=	0.84375	0	.84375
.84375	*	2	=	1.6875	1	.6875
.6875	*	2	=	1.375	1	.375
.375	*	2	=	0.75	0	.75
.75	*	2	=	1.5	1	.5
.5	*	2	=	1.0	1	.0

Therefore, $-2337.7109375_{10} = -100100100001.1011011_2$. Again, we can fill the floating point table. First, the number is negative; therefore, the sign bit is 1:

+	Exponent								Mantissa															
-																								
1																								

Next, the mantissa is 1001001000011011011 (we shifted the decimal point all the way to the right):

+	Exponent								Mantissa																						
-																															
1									0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	1	1	0	1	1

Finally, the exponent is -7 (since the decimal point needs to be moved seven places to the left). We convert -7 to binary using two's complement notation: $00000111 \rightarrow 11111000 \rightarrow 11111001_2$:

+	Exponent								Mantissa																						
-																															
1	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	1	1	0	1	1

Again, we can recreate the decimal representation of the number. We know that the number is negative because the sign bit is 1. We can convert the exponent: $11111001 \rightarrow 00000110 \rightarrow 00000111_2 = -7$. And we can convert the mantissa: $1001001000011011011_2 = 299227_{10}$. We then combine the components:

$$-299227 * 2^{-7} = -299227 * \frac{1}{2^7} = -299227 * \frac{1}{128} = -299227 * 0.0078125 = -2337.7109375$$

Did you know?

It is interesting to note that some fractions that can be expressed precisely in decimal notation, such as $1/10 = 0.1$, do not have an exact floating point representation. This is due to the fact that floating point numbers are represented using base two. One-tenth can be approximated in binary as 0.0001100110011001... (where 1001 repeats indefinitely). It can never be represented exactly. This result should not be surprising. After all, many fractions (e.g., $1/3$) cannot be represented exactly as decimal values. This is one reason why computed results that involve fractions are not always 100% accurate. They sometimes suffer from round off error as a result of the base ten to base two conversion process.

Combinational circuits

Recall that combinational circuits are digital circuits that do not involve any kind of feedback. In other words, the output of a combinational circuit cannot be fed back into that circuit as input. Previously, only three types of combinational circuits were covered: the *xor* gate, comparators (for equality, less than, and greater than), and adders (half adder, full adder, chaining full adders). There are several more types of combinational circuits that are important, particularly in the design of computers.

Decoders and encoders

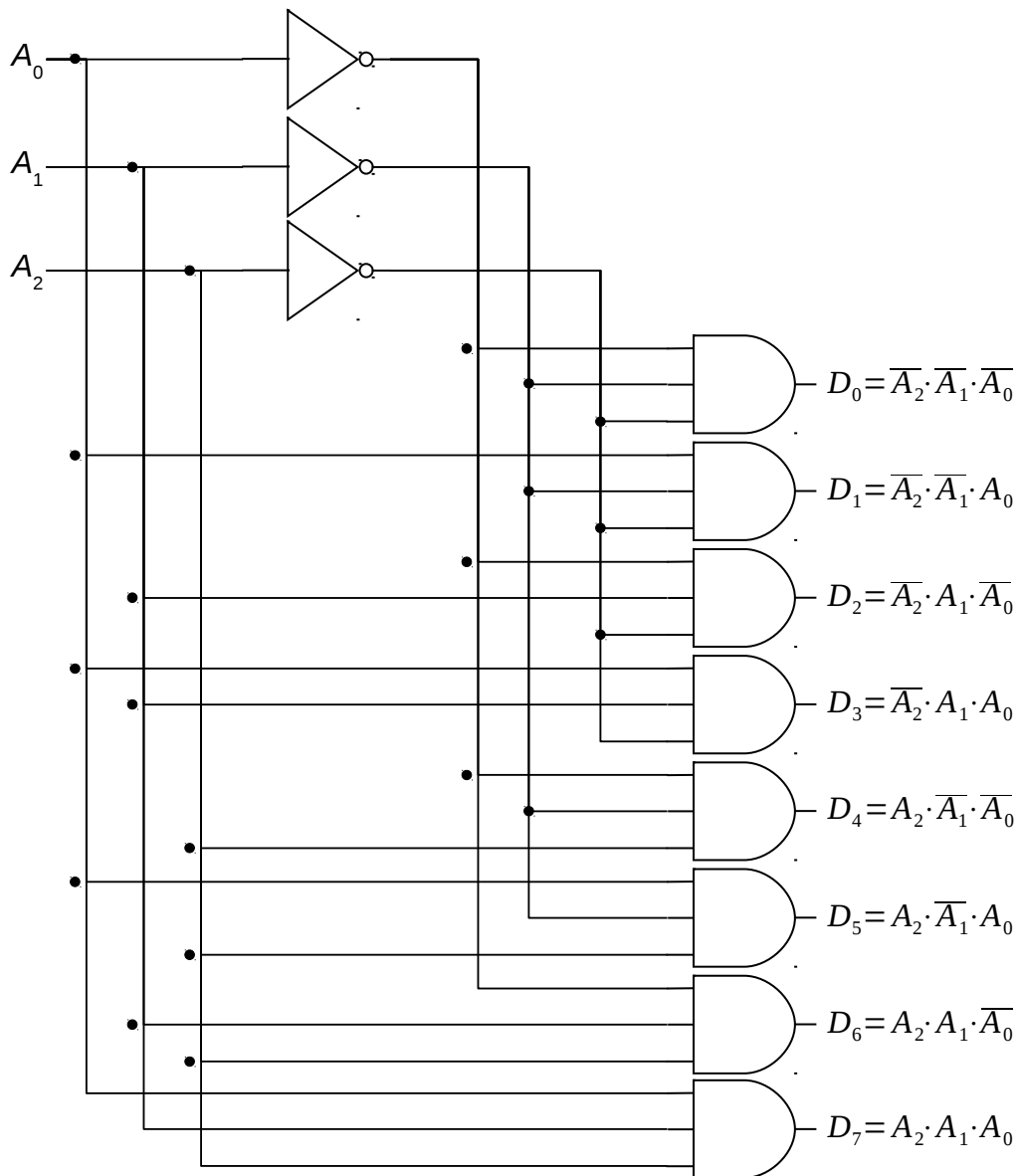
A **decoder** is a type of circuit that takes in a number (in unsigned binary form) and generates a 1 (high) on the output line that corresponds to the input number. All other output lines are set to 0 (low). For example, given an input of 00_2 , a 1 would be generated on output line zero. Likewise, given 01_2 as input, a 1 would be placed on output line one. You will see later how decoders form an integral part of memory and register addressing circuitry.

Every decoder with n input lines will have exactly 2^n output lines. Therefore, a *two-to-four* decoder will have *two* input lines and *four* output lines, while a *three-to-eight* decoder will have *three* input lines and *eight* output lines. Both the input and output lines of decoders are numbered, with the n inputs ranging from 0 to $n-1$, and the 2^n outputs ranging from 0 to $2^n - 1$. Here is the truth table for a three-to-eight decoder:

A ₂	A ₁	A ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

The inputs are labeled A₀ to A₂ and represent the bits of a three-bit unsigned binary number. The outputs are labeled D₀ through D₇. As the table shows, an input number (such as $110_2 = 6_{10}$) results in the corresponding data line (D₆ in this case) being set to 1, with all other lines held at 0.

The following circuit illustrates an implementation of a three-to-eight decoder. The circuit diagram uses eight three-input *and* gates (one for each data line), together with a total of three *not* gates. If you don't happen to have access to three-input *and* gates, remember that they can easily be constructed from two standard two-input *and* gates.

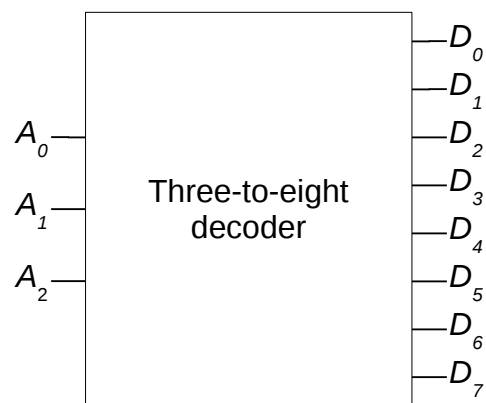


Carefully tracing the lines in the circuit diagram, we see that all three of the inputs to the D_0 *and* gate are negated. Thus, when inputs A_2 , A_1 , and A_0 each have a value of 0, D_0 's three-input *and* gate receives three 1s (i.e., *not* A_2 , *not* A_1 , and *not* A_0) and generates a 1 as output. The result is that a 1 (high) is placed on output line zero when the number 000_2 is given as input to the circuit.

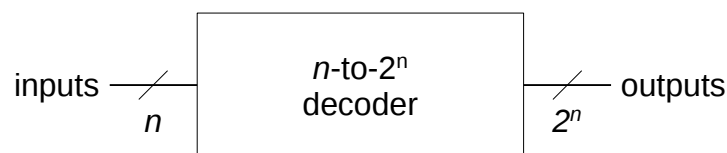
Let's continue to the *and* gate for D_1 . Note that two of the three inputs (A_2 and A_1) are negated, but the third input (A_0) is not. Thus, when inputs A_2 and A_1 are 0, but A_0 is 1 (corresponding to the input number 001_2 – or 1_{10}) the *and* gate will receive three 1's and generate a 1 on line D_1 . Skipping ahead to the final case, note that the three-input *and* gate for D_7 receives all of its inputs directly from A_2 , A_1 , and A_0

without negation. Thus, when A_2 , A_1 , and A_0 each contain 1 (corresponding to the number 111_2 – or 7_{10}) the gate will generate a 1 on data line D_7 . The other cases are handled in a similar manner.

The three-to-eight decoder may be encapsulated using a “black box” (as you've seen before) as follows:



In general, since an n input decoder will always have 2^n outputs, its “black box” can be expressed in the following way:



The following symbol (along with a number or variable), is frequently used as a shorthand in circuit design to represent the indicated number of lines without actually drawing them. Since modern computers are based on 32-bit buses, this compact representation is very important.



It is natural to wonder at this point if there is a circuit that does the exact opposite of a decoder. Of course there is! An **encoder** is a type of circuit with 2^n input lines, only one of which can be high at any given time, that produces an n -bit unsigned binary number that corresponds to the raised input line.

A four-to-two encoder is defined by the following truth table:

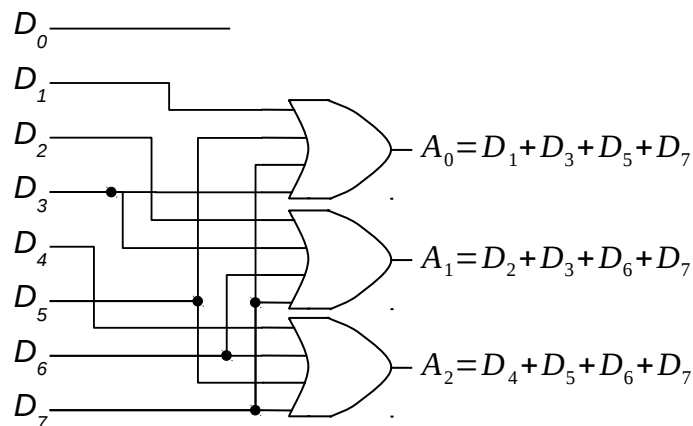
D_3	D_2	D_1	D_0	A_1	A_0
0	0	0	0	?	?
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	?	?
0	1	0	0	1	0
0	1	0	1	?	?
0	1	1	0	?	?

D_3	D_2	D_1	D_0	A_1	A_0
0	1	1	1	?	?
1	0	0	0	1	1
1	0	0	1	?	?
1	0	1	0	?	?
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?

Note that twelve of the sixteen rows of the truth table are invalid. These rows represent input configurations that are disallowed because either no input line is high, or multiple input lines are high. In these cases, there is no single high input line for the circuit to return the corresponding binary number of.

How are we to deal with these *undefined* configurations when constructing the corresponding circuit? One way is to simply ignore the disallowed input configurations. This approach will work fine as long as we can be *sure* that the illegal states can never occur.

The following presents an implementation of an eight-to-three encoder that assumes that disallowed states will never be encountered. The circuit works fine as long as this limitation is respected. For example, placing a 1 on line D_6 , while holding all other lines low, causes the number 110_2 (or 6_{10}), to be generated. An invalid configuration of inputs generates an erroneous output. For example, setting both D_1 and D_2 to 1 causes the circuit to output 011_2 (or 3_{10}).

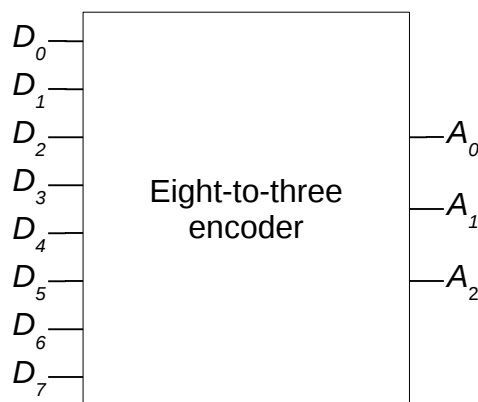


One odd feature of this circuit is that input line D_0 is not connected to anything. It is, in a sense, ignored. Although this may seem strange at first, it is precisely what we want the circuit to do. Setting line D_0 to 1 (and all other input lines to 0) is supposed to cause all of the output lines to be set to 0 in order to represent the number 000_2 .

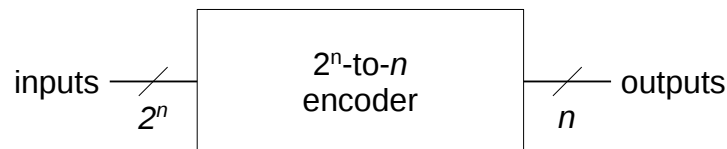
The truth table for the eight-to-three encoder would be similar to the one shown above for the four-to-two encoder. However, the full truth table would consist of 256 rows (since it has eight input lines and therefore $2^8 = 256$ possible input configurations). All but eight of these input configurations would be disallowed. In order for the circuit's truth table to fit on a page, only the allowed rows are presented below:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	A ₂	A ₁	A ₀
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

As you can see, this truth table is the exact inverse of the truth table for the three-to-eight decoder that was presented earlier. The eight-to-three encoder can be encapsulated into a “black box” as follows:



In general, a 2^n to n encoder can be drawn as follows:



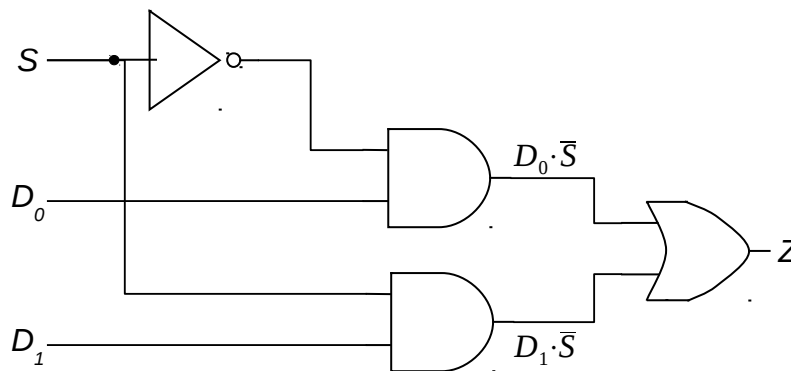
Multiplexers and demultiplexers

Multiplexers are circuits that are used to transfer the contents of an input data line to an output line based on the value of one or more input selector lines. While that definition sounds a bit intimidating, multiplexers really aren't that complex. Essentially, a multiplexer is a kind of switch. It has two types of inputs: data lines and selector lines; it has a single output line. The purpose of a multiplexer is to transfer the contents of *one* of the input lines to the circuit's single output line. The values placed on the selector lines determine which data line will have its contents transferred to the output line. Thus, the

selector lines allow the multiplexer circuit to switch its “attention” between the various input data lines when determining the value to be output.

Generally, a multiplexer (MUX) will have 2^n data lines for n selector lines. The data lines are numbered 0 to $2^n - 1$, and the selector lines are numbered 0 to $n - 1$. The bit pattern placed on the selector lines, when interpreted as an unsigned binary number, determines the active data line. Multiplexers are often referred to in terms of their number of data lines. Therefore, an eight-input MUX will have eight data lines and three selector lines, while a two-input MUX will have two data lines and a single selector line. Remember that, regardless of the number of input data and selector lines, every MUX has only one output line.

The following presents an implementation of a two-input multiplexer. It has two data lines, one selector line, and a single output. When the selector line, S , is set to 0, the current value of D_0 (either a 1 or a 0) becomes the output of the circuit, regardless of the value of line D_1 . The circuit is, in a sense, *listening* to D_0 and ignoring D_1 . When S is 1, the opposite situation exists: the output of the multiplexer becomes the current value of D_1 , and D_0 is ignored.

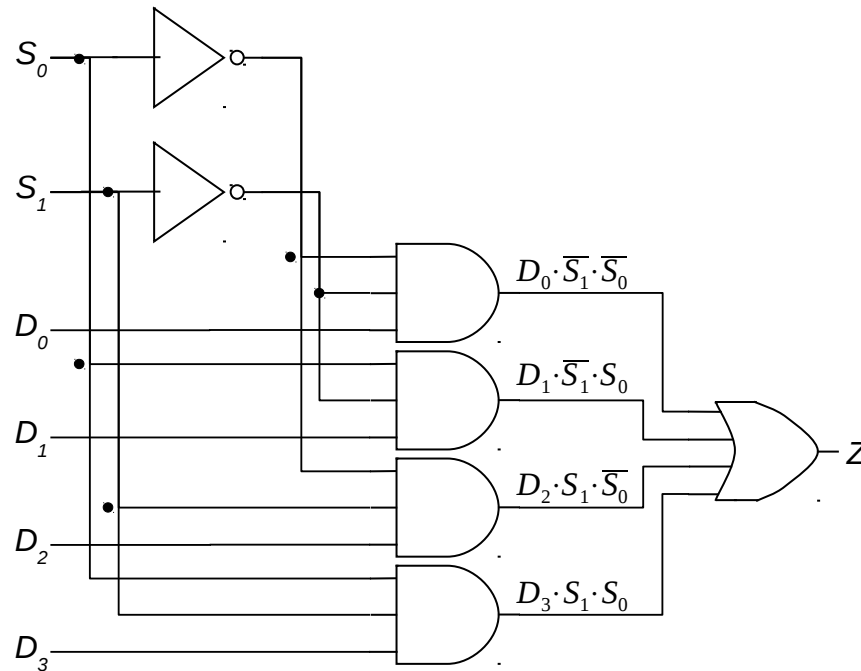


The behavior of the two-input multiplexer is summarized in the following truth table:

S	D ₁	D ₀	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The following illustrates a somewhat more complex multiplexer, a four-input MUX. It has two selector lines, four data lines, and a single output line. Placing a 0 on both S_1 and S_0 , corresponding to 00_2 , causes the value of D_0 to be transferred to the output line. Setting S_1 to 0 and S_0 to 1, corresponding to 01_2 , causes D_1 to be transferred to the output line. Likewise, setting S_1 to 1 and S_0 to 0, corresponding to

10_2 , causes D_2 to be transferred to the output line. Finally, setting S_1 to 1 and S_0 to 1, corresponding to 11_2 , causes D_3 to be transferred to the output line:



An inspection of the circuit diagram for the four-input MUX reveals that it contains four three-input *and* gates, two single-input *not* gates, and one four-input *or* gate. Each *and* has one of the data lines running into it, plus two selector signals. Some of the selector signals have been routed directly from the selector input lines, while others have been negated before being sent on to the *and* gates. The outputs of all four of the *and* gates are routed into the four-input *or*. The output of this *or* becomes the output of the MUX circuit. If any one of the *and* gates generate a 1, the circuit will output a 1. If all of the *and* gates produce 0, the circuit will output a 0.

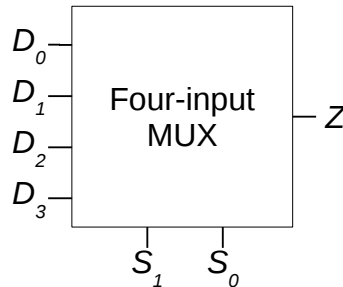
In order to better understand the behavior of this multiplexer, let's examine the conditions under which each of the *and* gates could fire. A three-input *and* gate can produce a 1 only if all of its inputs are 1. Thus, the two selector signals and the data value reaching the *and* gate must be 1 for that gate to generate a 1.

The selector signals reaching the gate that is connected to D_0 are *not* S_1 and *not* S_0 . This gate can produce a 1 only when $S_1 = 0$, $S_0 = 0$ and $D_0 = 1$. Under any other circumstances (e.g., if D_0 is 0 or if either S_1 or S_0 is 1) this *and* gate produces a 0. Hence, this part of the circuit faithfully transfers the value of D_0 when the S_1, S_0 bit pattern is 00. Just as importantly, this *and* gate stays low when the selector bit pattern is not 00, regardless of the value of D_0 .

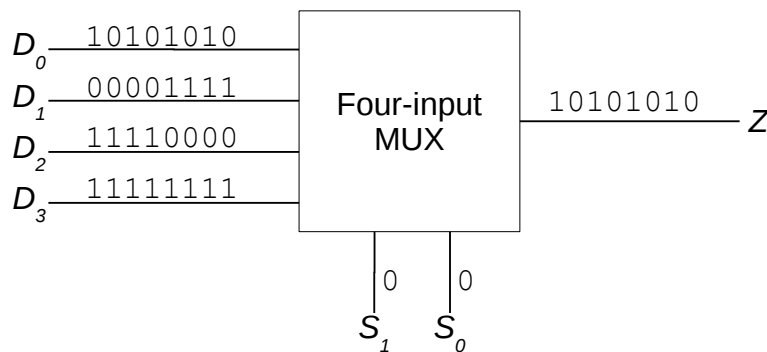
The other *and* gates act in a similar manner. The gate for D_1 is attached to *not* S_1 and S_0 ; therefore, it only generates a 1 when $S_1 = 0$, $S_0 = 1$, and $D_1 = 1$. Other selector bit patterns keep it low. The *and* gate that receives the D_2 signal is connected to S_1 and *not* S_0 . This gate produces a 1 only when $S_1 = 1$, $S_0 = 0$, and $D_2 = 1$. It generates a 0 at all other times. Finally, the *and* gate for D_3 is attached directly to S_1 and S_0 . Thus, it generates a 1 when $S_1 = 1$, $S_0 = 1$, and $D_3 = 1$.

Because of the way the selector signals are routed to the various *and* gates, it is impossible for more than one of them to produce a 1 at the same time. For this reason, the results of the *and* gates can be safely combined via an *or* gate without worrying that signals from multiple data lines will be accidentally combined.

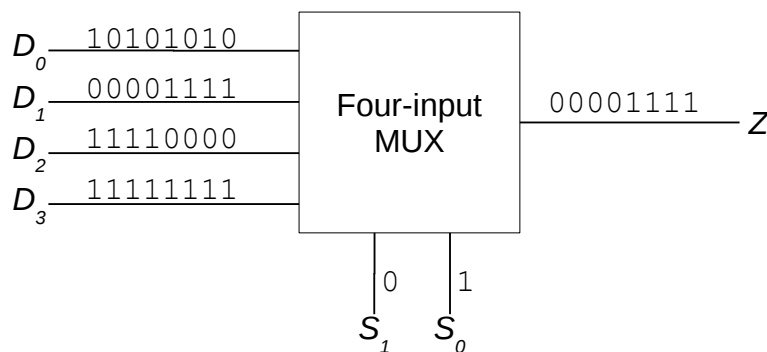
The four-input multiplexer can be represented by a “black box” as follows:



The following figures illustrate the behavior of the four-input MUX over time. The input lines and output lines are labeled with the data streams flowing down them. During the period of time illustrated in the following figure, selector lines S_1 and S_0 are both held at zero, making D_0 the *active* data channel:



In the following figure, S_1 is held at zero and S_0 at one, thereby activating D_1 :

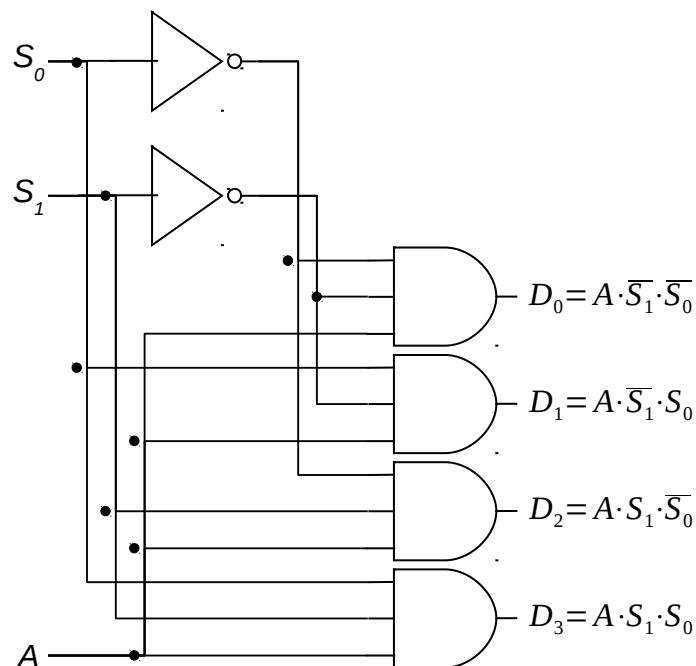


While the selector lines are held steady, the current state of each of the data lines varies over time as data moves across the lines (from left-to-right; therefore, the data on the lines is read from right-to-left). For example, in both of the previous figures, line D_0 first contains a 0, then its value changes to 1, then back to 0, then to 1, then 0, then 1, and so on. Line D_1 begins by broadcasting four consecutive 1s followed by four consecutive 0s.

Notice that when D_0 is active (i.e., $S_1 = 0, S_0 = 0$), its bit pattern, 01010101, is copied to the output channel. Likewise, when D_1 is active (i.e., $S_1 = 0, S_0 = 1$), its bit pattern, 11110000, is copied to the output channel.

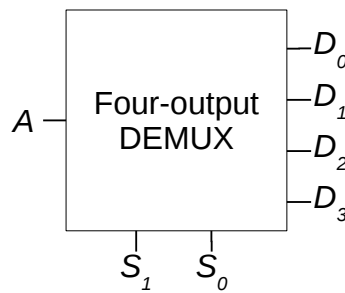
Just as with the decoder above, the multiplexer has an exact opposite circuit: the demultiplexer (DEMUX). Each demultiplexer has a single data input line, n selector input lines, and 2^n output data lines. As was the case with multiplexers, the n selector lines are numbered from 0 to $n - 1$, and the 2^n output data lines from 0 to $2^n - 1$. Demultiplexers generate a copy of their input data value on the output data line specified by their selector lines.

Demultiplexers are often referred to using their number of output lines. The following illustrates an implementation of a four-output demultiplexer:

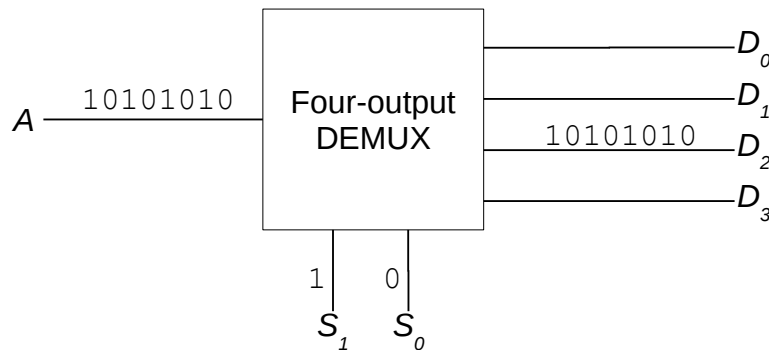


The value of the input data line labeled A is transferred to one of the output data lines, D_0 through D_3 , based on an interpretation of the bit pattern in S_1, S_0 as a two-bit unsigned binary number. For example, if $S_1 = 1$ and $S_0 = 0$, corresponding to 10_2 (or 2_{10}), then the current state of the input line would be transferred to D_2 . The design of this circuit is quite similar to the one used for the decoder circuit shown earlier. The only difference is that a copy of the input data value is routed to each of the *and* gates so that, instead of simply setting the selected output line high, its value will instead be determined by the input data value.

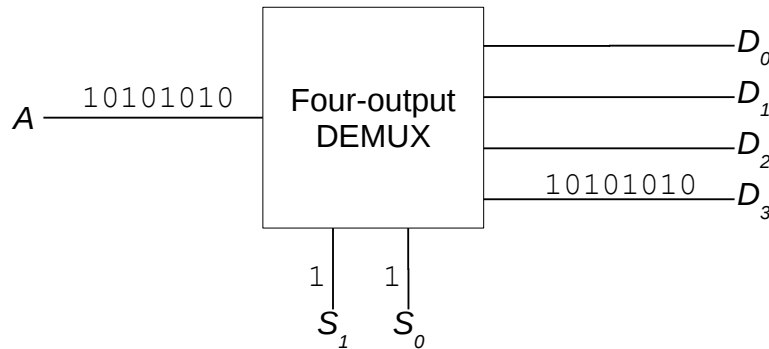
The following is a “black box” representation of the four-output demultiplexer:



The following figures illustrate the behavior of the four-input DEMUX over time. The input lines and output lines are labeled with the data streams flowing down them. During the period of time illustrated in the following figure, selector line S_1 is held at one and S_0 is held at zero, making D_2 the active data channel.



In the following figure, S_1 and S_0 are both held at one, activating D_3 :



These examples illustrate how a demultiplexer can *broadcast* an input data stream down one of many different output channels. Changing S_1, S_0 changes the broadcast to a different output channel.

Sequential circuits

All of the circuits discussed up to this point have been combinational circuits whose outputs depended solely on their inputs. Such circuits do not incorporate feedback and have no “memory” of their previous state. In order to construct a general-purpose computer, circuits capable of remembering instructions, data, and the results of computations are needed.

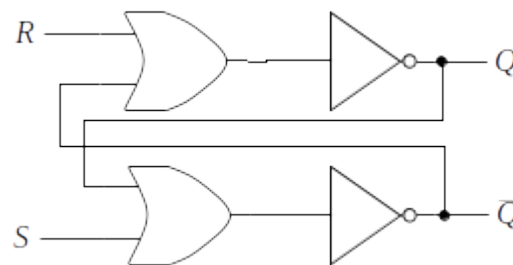
It is possible to design circuits that exhibit memory by incorporating feedback, in the sense that the outputs of a circuit can be made to depend not only on the circuit’s current inputs, but also on its past outputs as well. Such circuits are referred to as **sequential circuits**, since their output may be viewed as

a function of a sequence of past inputs. Basically, sequential circuits have memory because one or more of their outputs are *fed back* to serve as input. Therefore, a sequential circuit's next output will, in a sense, be a function of its present inputs and its previous outputs.

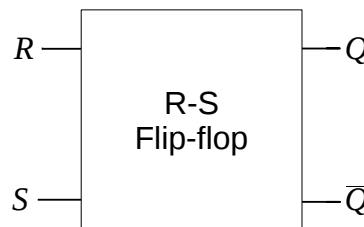
We will limit our study of sequential circuits to one, very important, type of circuit: the flip-flop. The term **flip-flop** is a generic term applied to devices having two stable states. The primary function of a flip-flop is to store a binary digit, 0 or 1. Therefore, a flip-flop can be used to implement the most basic unit of memory, the bit. A flip-flop is implemented as a set of logic gates that make use of feedback to remain in one of two stable states, thereby *remembering* a binary digit.

The R-S flip-flop

An R-S flip-flop can be constructed from two interconnected *nor* gates. The following figure illustrates such an R-S flip-flop. As discussed in a previous lesson, a *nor* gate is simply an *or* gate followed by a *not* gate. For clarity, the following figures shows the two *nor* gates of the flip-flop as having been decomposed into their underlying *or* and *not* gates:



Here's a “black box” representation of the R-S flip-flop:

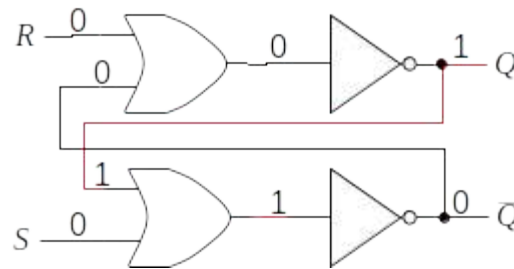


R-S flip-flops have two inputs and two outputs. The inputs are usually labeled R for **Reset** and S for **Set**. The outputs are traditionally labeled Q and \bar{Q} (*not* Q). \bar{Q} is the complement (or opposite) of Q; therefore, if Q is 1 then \bar{Q} should be 0 (and vice versa).

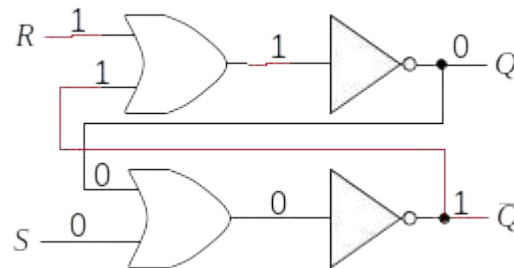
The Q output of the flip-flop determines its state. In other words, examining the Q output is the same as seeing what is stored in the bit. The state can be a binary 1 (Set) or a binary 0 (Reset). When a flip-flop like the R-S is in one of its states (Set or Reset), it will remain unchanged until an appropriate signal or pulse is applied to one of its input lines. The ability of a flip-flop to hold the Q output constant until a signal is given to change the value of Q is why flip-flops are considered basic memory devices.

We begin our detailed analysis of the R-S flip-flop assuming that Q is initially high (1), and both R and S are low (0), as shown in the following figure. Since both of the inputs to the top *or* gate are low, its output is low, and the output of the top *not* gate (the Q signal) is high. This high output is fed back into the input of the bottom *or* gate making its output high and the output of the *not* gate (the \bar{Q} signal) low.

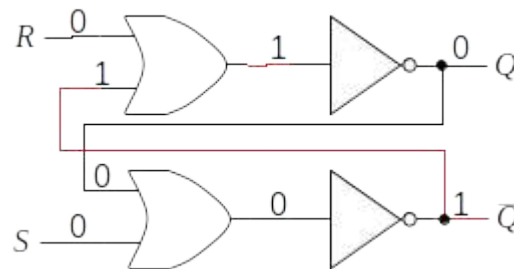
This is a stable circuit. While R and S remain low, Q will remain high and \bar{Q} low. The bit is thus holding a 1.



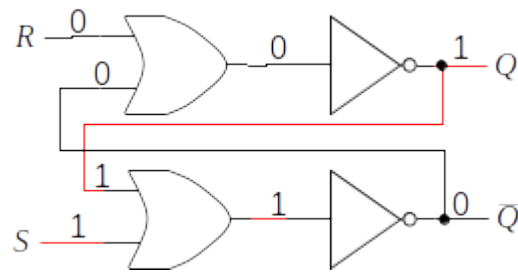
If we apply a 1 to R in order to place a 0 into the bit (Q), the following events occur: (1) the output of the top *nor* gate (the Q signal) goes low; (2) this is fed back to the input of the bottom *nor* gate, which causes its output (the \bar{Q} signal) to go high; and (3) this signal is fed back to the top *nor* gate, but since it already has another high input, there is no change in the output (i.e., it remains low). This is a stable circuit, as shown in the following figure (note that the flip-flop has been reset to 0):



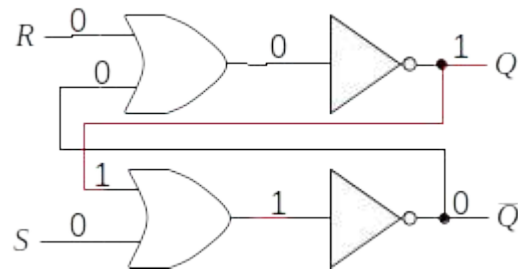
If R is set to low, the flip-flop remains in the Reset (0) state since there is still a 1 input to the top *nor* gate from the \bar{Q} input. This is shown in the following figure. Note that this is exactly the behavior we desire. In order for the flip-flop to be useful, it must be able to *remember* that it has been reset to 0 even after the reset signal is removed.



Next, we apply a high signal to the S input in order to set the bit to 1. This action causes the following events to occur: (1) the output of the bottom *nor* gate becomes 0 since one of its inputs is now high; (2) this low signal is applied as an input to the top *nor* gate, where the R input is also low; and (3) the output of the top *nor* is forced high, and this high output is fed back into the bottom *nor* gate, which does not change its state (since its other input was already high). The flip-flop has been set to 1, as can be seen in the following figure:

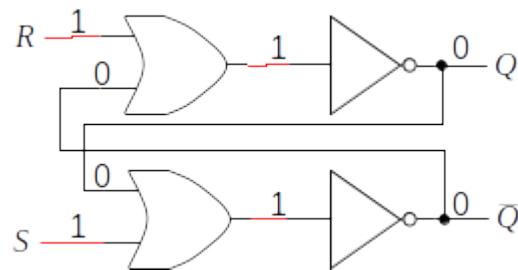


Next, we remove the Set signal. In other words, we change the S input to low. This action does not change the output of the bottom *nor* gate since its other input is already high and the circuit remains in the set state. This is shown in the following figure. Thus, once an R-S flip-flop receives a pulse (or 1) down its set line, it will continue to hold that 1 until a reset signal (i.e., 1 down the reset line) is received.



Notice that state of the flip-flop shown above is identical to its original state (shown at the very beginning of these examples). They both represent the flip-flop in its 1 state, with 0 on both input lines. Comparing the figures to one another illustrates another feature of flip-flops: the value output by the circuit is not solely dependent on its current inputs. When the R and S inputs are both 0, the value output on the Q line depends on whether the most recent 1 input was on the set line or the reset line.

There is one other possible configuration of inputs we have not yet considered. What happens if both the R and S inputs are set to high at the same time (corresponding to an attempt to simultaneously store both a 0 and a 1 into a single bit)? This is pictured in the following figure. The outputs do remain constant, but the R-S flip-flop is not in a valid state because Q and \bar{Q} have identical values, yet they are always supposed to be the opposite of one another.



To conclude the introduction to the R-S flip-flop, here is its truth table:

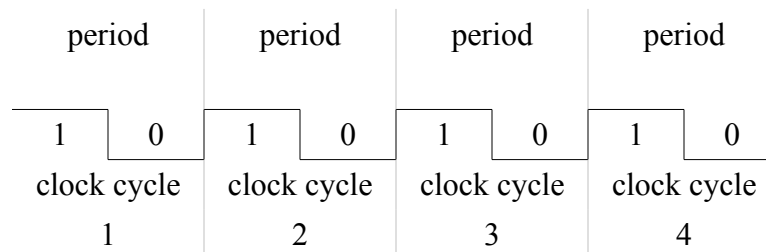
S	R	Q	\bar{Q}
0	0	no change	
0	1	0	1

1	0	1	0
1	1	disallowed	

The clocked R-S flip-flop

A **clock** is a device that generates a signal that periodically cycles between a high state and a low state. Clocks ensure that the operations performed by a computer proceed in an orderly manner. They do so by enabling certain operations to occur only at specific points in time.

Clocks divide time into **cycles** that consist of two phases: a high phase and a low phase. Specifically, a **clock cycle** is defined as the interval of time beginning when the clock goes to a high state, lasting through the return to a low state, and ending with the start of the transition back to the high state again. the following figure illustrates four complete cycles of a clock:

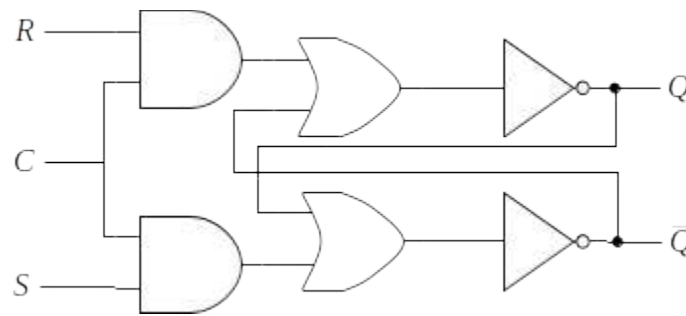


Each clock cycle lasts for only a brief instant of time. The CPU of a modern PC, for example, runs at billions of clock cycles per second (or gigahertz). The clock speeds of other components, such as the system bus, are usually somewhat slower but still in the range of hundreds of millions of clock cycles per second (or megahertz). The various operations that a computer can perform require one or more clock cycles to complete. The exact number of cycles depends upon the complexity of the particular operation.

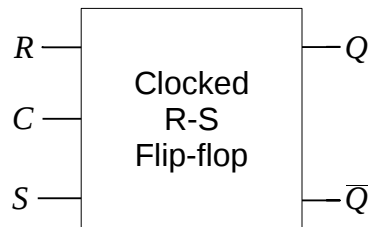
As mentioned earlier, flip-flops can be used to implement the most basic unit of storage, the bit. Memory devices based on R-S flip-flops perform read operations by retrieving the contents of the Q outputs of a number of selected bits. Similarly, the write operation stores bit patterns into memory by placing 1s on either the S (Set) or R (Reset) inputs of various bits.

The clock ensures that these operations happen in an orderly manner. During one phase of the clock cycle (e.g., low) the contents of memory can be examined but not modified. During the opposite phase of the cycle (e.g., high) the contents of memory can be updated. This sort of timing is critical for the reliable operation of a computer because it allows time after a *write* operation for the flip-flops to settle into their stable configurations before *read* operations can be attempted.

The following figure presents the circuit diagram of a clocked R-S flip-flop. In addition to the R and S inputs, these circuits also receive the clock signal. In the clocked R-S flip-flop, the Q output will be unaffected by any change in R or S as long as the clock (C) is 0. That is, during the *read* phase of the clock cycle, the contents of memory cannot be changed. When the clock input goes to 1, designating the *write* phase of the clock cycle, the Q output will change depending upon the values of R and S.



Lastly, the following is a “black box” representation of the clocked R-S flip-flop:

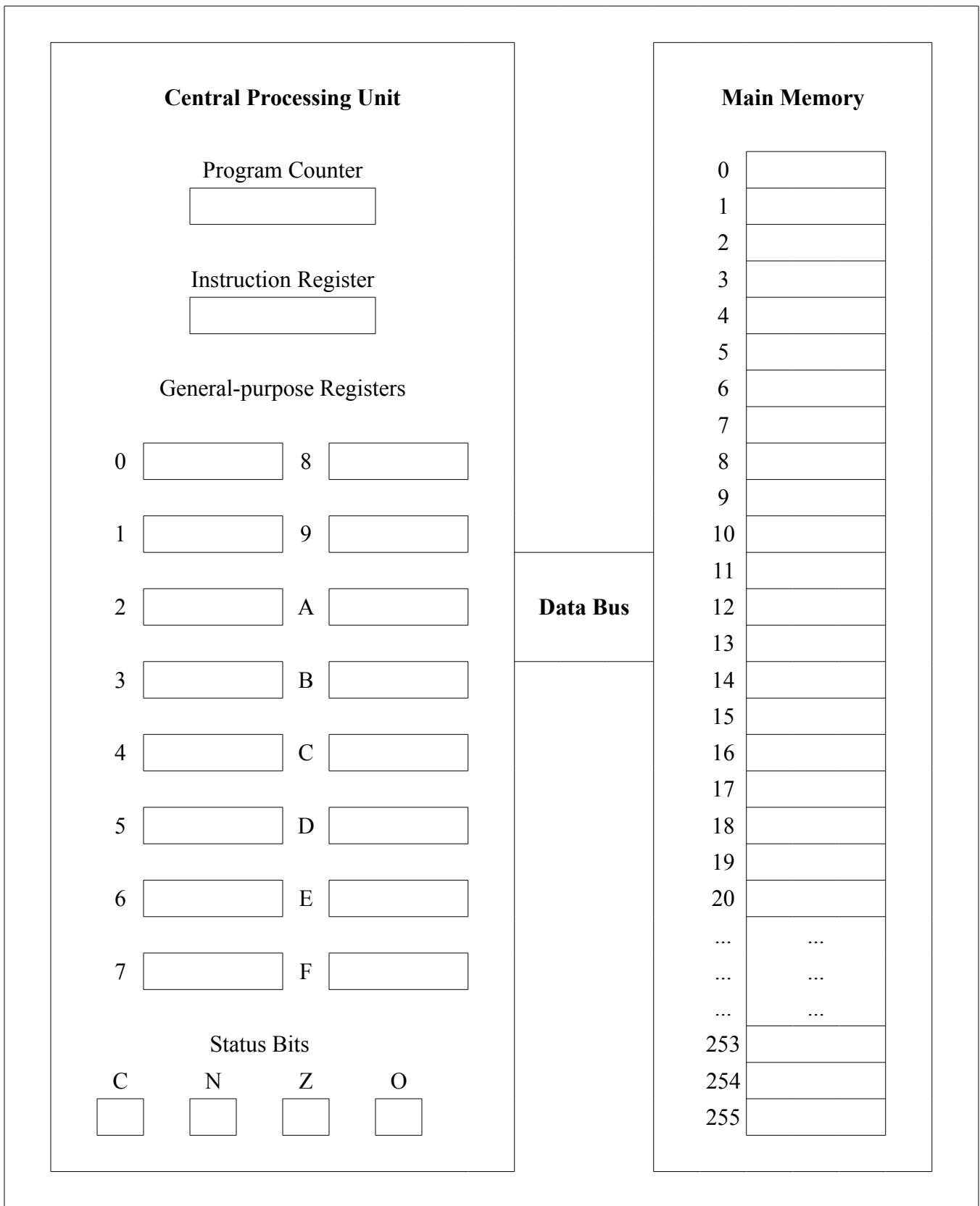


Building a simple computer

At this point, we have all of the building blocks required to design a simple microcomputer. Primarily, we will focus on two of the higher-level components, the arithmetic logic unit (ALU) and main memory, and how they can be constructed from low-level circuits. In order to have a context in which to frame this discussion, we need to briefly discuss register-based machines.

The general organization of a machine is often referred to as its **architecture**. Over the years, computer scientists and engineers have explored a wide variety of computer architectures. The register-based machine is the most popular architecture. A **register** is similar to memory in that it can store data; however, it is much more quickly accessible because it is located within the CPU.

Although we will use a “toy” computer, designed solely for the purpose of instruction, it embodies most of the major features of register-based machines. The sample (virtual) machine that we will use is shown in the figure below. It consists of two major parts: main memory and a CPU. These two components are connected together via the data bus, which allows information to be copied between the CPU and main memory.



The purpose of main memory is to store computer programs and the data on which they act. The main memory of the virtual machine can be visualized as a list (or array) of 256 storage locations, numbered 0 to 255. Each of these locations is individually addressable. In other words, the main memory unit can be given an address and told to retrieve a value from that location, or be given an address and told to write a value to that location.

The purpose of the CPU is to perform the arithmetic and logic functions specified by the instructions of a computer program. CPUs are complex devices composed of many functional units. One of the most prominent features of the CPUs of register-based machines is a large collection of general-purpose registers. The virtual machine contains sixteen registers, labeled 0 through 9 and A through F.

Registers can be directly manipulated by the arithmetic and logic units of the CPU. Main memory locations cannot. In order to perform any kind of arithmetic or logic operation on values stored in main memory, it is necessary to copy those values into CPU registers, manipulate the contents of the registers in a desired manner, and then copy the computed results back to main memory. For example, in order to add two numbers stored in main memory locations, the virtual machine must:

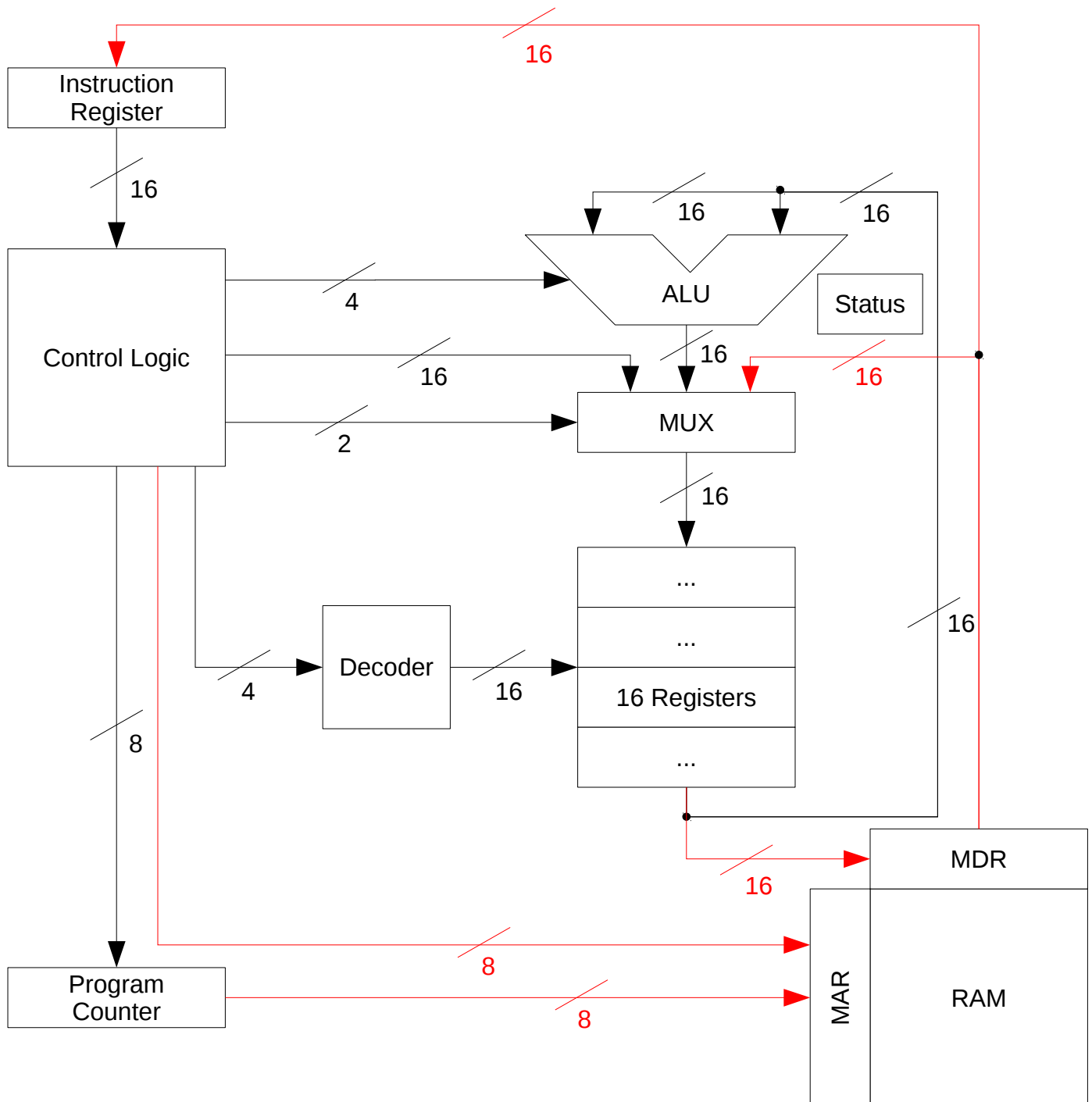
- (1) Copy the values of both numbers from main memory into separate CPU registers;
- (2) Add the contents of those registers, placing the result into yet another register; and
- (3) Copy the result of the addition operation back into a main memory location.

In addition to the general-purpose registers, there are a number of special-purpose registers important to the operation of the CPU. These registers include the instruction register, the program counter, and the status bits. The **instruction register** holds a copy of the program instruction that the CPU is currently executing. The **program counter** contains the address of the next instruction to be executed. The **status bits** are used to hold information about the most recently performed computation (e.g., was a carry generated from some binary addition, was the result zero, was some result negative, did an overflow occur, and so on). More detail about these special-purpose registers is beyond the scope of this lesson.

Real world computers are similar to the virtual machine in that they contain main memory, a CPU, and a data bus. The virtual machine differs from real machines in that it has no I/O devices (such as keyboards, display screens, mice, and so on), nor any long-term storage devices (such as hard disk drives). The size of its memory is also very small, containing only 256 memory locations. Modern computers typically have hundreds of millions of memory locations. Technically, these memory locations are known as words. A **word** is the base unit of data that is supported by a CPU. Today's computers usually have word sizes that are 32 or 64 bits. The registers are typically word-sized. There are other differences between the virtual machine and real-world computers, such as the size of the numbers it can manipulate and the limited number of commands in its machine language.

The main memory of the virtual machine is composed of 256 words, each 16 bits wide. The interface to main memory (RAM) consists of two registers: the memory address register (MAR), and the memory data register (MDR). During both “read” and “write” operations, the MAR is used to hold the address of the memory location to be accessed. Since there are 256 words of memory in the virtual machine (numbered 0 to 255), the MAR is eight bits wide, enabling it to hold addresses in the range 00000000_2 to 11111111_2 (i.e., 0 to 255). During “write” operations, the 16-bit value to be written into memory will be placed in the MDR. During “read” operations the value of the memory location to be read will be output to the MDR. Thus, the memory data register can function both in an input role (for “write” operations) and in an output role (for “read” operations). The MAR, on the other hand, always functions in an input role, specifying the location to be accessed.

The following figure primarily details the virtual machine CPU. Special emphasis is given to illustrating the major control and data lines interconnecting the various components:



We now address the final piece of the puzzle concerning machine language programs by illustrating how the machine actually goes about executing a program. At the machine level, all a computer ever does is perform the following five tasks (collectively known as the **instruction cycle**) over and over:

- (1) Fetch the next instruction from memory. To do so, load into the instruction register the bit pattern found at the address held in the program counter.
- (2) Increment the program counter by one so that it points to the next instruction in the current sequence.
- (3) Decode the current instruction. This involves correctly identifying the various parts of an instruction based on its op-code. The **op-code** is the part of a machine language instruction that defines what operation to perform (e.g., addition). The other parts of a machine language instruction are called **operands** on which the op-code is applied.
- (4) Execute the instruction. Operand values are routed to the appropriate arithmetic and logic hardware. Results are computed and routed to their appropriate destinations. Appropriate destinations include the general-purpose registers, special purpose registers such as the program counter, and main memory.
- (5) Return to Step 1.

That's all a computer ever does! There is no magic; no smoke and mirrors. Just a simple, but very fast machine running through the instruction cycle over and over, tens or hundreds of millions of times each second.

Virtual machine overview

The control logic, or control unit, is the component of the virtual machine that is responsible for implementing the instruction cycle. It does so by generating the signals necessary to direct the other components of the machine to carry out their tasks in an orderly manner. For this reason, the control unit is sometimes referred to as the “traffic cop” of the CPU. Input to the control unit is from the instruction register, since it must have access to the bit pattern that represents the instruction for it to do its job. Because of the central role played by the control unit, its outputs are connected by various data and signal lines to most of the other components of the machine.

The ALU is responsible for the math and logic operations performed by the machine. If the control unit is the “traffic cop”, then the ALU is the “calculator”, responsible for performing addition, subtraction, and the various logic operations. The ALU of the virtual machine receives two 16-bit input values from the general-purpose registers and one 4-bit control code from the control unit. The 16-bit values represent the operands. The control code, derived from the op-code of the current instruction, specifies which arithmetic or logic operation is to be performed on those operands. Output from the ALU is directed to a multiplexer that is responsible for routing data to the general-purpose registers. This makes sense, because the results of arithmetic and logic operations must end up in one of the registers.

Taking a closer look at the multiplexer, we see that it has three 16-bit data inputs, one from the control unit, one from the ALU, and one from the MDR. There is also a two-bit selector signal sent from the control unit to the multiplexer in order to tell it which input data values should be passed on.

The reason for the three data inputs into the multiplexer has to do with the sources that can generate register values. Arithmetic and logic instructions (like addition), require that the registers be able to receive input from the ALU in order to place the result of the operation into the destination register. Other instructions require that registers be able to receive data from main memory or from “immediate” values, which are part of the instruction (and are therefore located in the instruction register). The two-bit selector signal, under the direction of the control unit, specifies which of these three sources should be routed to the registers.

Building this multiplexer is a straightforward exercise: we simply arrange sixteen of the four-input multiplexers shown earlier in parallel. The reason for using four-input multiplexers, even though we only have three input sources, is that multiplexers only come in powers of two: two input, four input, eight input, etc. Therefore, one of the inputs on each of the sixteen standard four-input multiplexers will be unused. While this may seem a tad wasteful, it won't cause any operational difficulties.

The reason for needing sixteen copies of the "standard" four-input multiplexers is that our data values are sixteen bits wide. Thus, each of the "standard" four-input multiplexers will pass only one bit of the 16-bit data value to an output line. By arranging sixteen of these "one-bit wide" multiplexers in parallel, we can build a multiplexer capable of routing all 16 bits of the selected data value simultaneously.

The final component of the CPU that we will look at is the four-input decoder, located between the control unit and the bank of general-purpose registers. The task of this decoder is to select one of the sixteen registers for access, either for receiving a value from the MUX or for sending a register value to main memory or the ALU. This decoder would be similar to the three-to-eight decoder shown earlier; however, it would be extended to handle four input lines and sixteen output lines.

In addition to the memory and CPU, the virtual machine also contains a data bus. This bus was illustrated earlier simply as a path connecting the CPU and memory. In the virtual machine, the representation of the data bus is much more detailed. It is presented as a number of separate data and address lines (highlighted in red in the figure above) that connect components of the CPU to RAM. In the figure above, there are two sets of 8-bit address lines, both leading into the MAR. There are also two sets of 16-bit data lines connected to the MDR, one set for input and one set for output.

The memory location to be retrieved (and therefore the address to be loaded into the MAR) can originate from two separate CPU components: the control unit or the program counter. The address comes from the program counter when the machine is fetching the next instruction from memory. The address comes from the control unit when the machine is fetching an operand, such as a variable, from memory.

The input data lines leading to the MDR originate from the block of sixteen general-purpose registers. This makes sense, because in order to store a value into memory, the value must first be in a register.

Looking at the output data lines originating from the MDR, we see that a 16-bit memory value can be transferred to either the instruction register or the multiplexer that routes values to the general-purpose registers. If the machine is performing an instruction fetch, the bit pattern being retrieved from memory represents an instruction and therefore must be sent to the instruction register. If the machine is in the process of fetching an operand, that operand should end up in one of the sixteen general-purpose registers and is thus sent to the multiplexer.

Even though the virtual machine is a very simple microcomputer by today's standards, it is still much too complex for us to go through a full design of its components. The development of its components, however, is constructed from the combinational and sequential circuits discussed throughout this curriculum. We have arrived at the bare machine that lies underneath the many layers of software. There is no magic; no smoke and mirrors.

The purpose of this lesson is to examine how ordinary people (or end-users), use computers to enhance communication and solve problems. The focus is on productivity or work related applications, rather than on recreational uses such as game playing (although those do serve an important purpose too).

The changing role of computers

Without computers modern life would be all but impossible. Computers keep track of our bank accounts, they power ATMs, generate our paychecks, print billing statements, and produce most of the “junk mail” we wade through. They enable us to purchase food, clothing, and other items with credit and debit cards. They allow monetary funds to be transferred, stocks to be traded, and airline reservations to be booked. They help engineers design and build airplanes, automobiles, roads, and bridges. They enable scientists to study distant planets, understand DNA, and model new drugs. In fact, computers have become so much a part of modern life that one wonders how engineering, business, or science was ever conducted without them.

In the not too distant past, only highly trained professionals used computers. Today, large numbers of people use computers on a daily basis for tasks such as email, instant messaging, web browsing, word processing, creating and giving presentations, spreadsheet modeling, and information storage and retrieval. In order to understand how we reached this point and, perhaps, where we are headed, it is useful to look back at how computing has changed the way we work.

Large mainframe computers transformed the way business was conducted in the late 1960s and 1970s, automating most record keeping, accounting, and billing operations. With the advent of personal computers and networks in the 1980s, businesses were transformed once again, becoming more decentralized and receptive to individual customer needs. By the 1990s, computers had become inexpensive and common enough for most employees to have immediate access to them, once again changing the way business was conducted. The first decade of the 21st century is witnessing the rise of wireless and mobile computing, extending network connectivity beyond the desktop and promising to once again fundamentally change the way we work and live.

Although the magnitude of these changes has been quite large, it is interesting to note that much of this change has occurred in unexpected directions, and that some expected changes have failed to materialize. For example, in the late 1970s, the Xerox Corporation funded advanced research into how computers would be used in business settings in the coming decades. The Altos research project conducted at PARC (Palo Alto Research Center) focused on developing prototype tools for the “paperless” office. The idea was that, when computers could be used for creating documents and networks for transmitting them, there would no longer be a need for paper in offices. This was, of course, a concern for Xerox since their major business was producing photocopiers.

While the Altos project led to many advances such as graphical user interfaces and mouse pointing devices that are in common use today, the anticipated “paperless” office never really arrived. In fact, with the widespread availability of high quality laser printers and the ability to quickly update manuscripts with word processors, paper consumption has increased rather than decreased.

Some people think that eventually paper will lose its dominant role as a medium for recording and disseminating information. They point out that computers are still relatively expensive, not yet

universally available, and suffer from a lack of software standards. Others believe that paper will continue to be cheaper and more portable for the foreseeable future.

Regardless of whether paper eventually goes the way of the dinosaur or not, one of the lessons of the Altos project is that progress in computing does not always proceed in easy to predict ways. As we take a look at the applications that are currently popular, you should try to keep this fact in mind. Even as recently as a decade ago, two of today's most popular end-user applications (web browsers and presentation software) didn't even exist. Thus, the applications you will be using ten years from now are likely to be very different from the ones that are popular today.

Communication-oriented applications

A communication-oriented application is one whose primary purpose is to enable humans to communicate with one another. The five most common applications are email, instant messaging, web browsers, word processors, and presentation software. All of these applications aim to enhance human communication. This is an amazing fact when one considers that computers were originally conceived of as "number crunchers" whose only function would be to perform complex calculations as rapidly as possible.

Email (or electronic mail) applications enable people to exchange text messages over networked computer systems. Email is a convenient and efficient way to communicate that has become vital to the conduct of business over the past decade. Email programs, such as Microsoft Outlook and Mozilla Thunderbird, integrate email functionality with related productivity applications such as address books, calendars, and scheduling applications.

Email can also provide a record of all messages sent to or from an account. While it is true that telephone conversations can be recorded, this is not usually done. Even if it were done regularly, the recordings would be difficult to search. Most email programs include searching, sorting, and archiving functions that make it relatively easy to track down a particular email message – even one exchanged months or years ago.

Another important feature of email is its ability to support attachments. Attachments are data files included in an email message. Some common attachments are: electronic copies of documents, images, sound files, and video. The ability to send copies of a document via email often leads to productivity gains in the workplace, since email is much faster and cheaper than alternative methods of transmitting documents, such as fax or photocopy and express mail.

In addition to facilitating one-on-one communication, email is also an excellent means for rapidly communicating with a group of people. Sending an email message to ten people is often just as simple as sending it to one. For this reason, email is generally the preferred way for members of a group to communicate with one another. Due to the ease of sending out messages to groups of people, physical paper-based memos are rapidly becoming a thing of the past in many offices.

A final aspect of email is the rise of web-based email (or web mail). Traditional email applications run on an end-user's computer. These programs connect to networked email servers in order to send outgoing email and download incoming email. In contrast, web-based email such as Google's Gmail is completely server based and accessed via a web browser.

Instant Messaging clients, or IM clients, support the sending and receiving of typed messages along with the exchange of data files. In this manner, IM clients are similar to email programs. However,

instant messaging systems also support *presence*, which means a user can see which of his or her IM contacts are online and the status of each (e.g., idle, away, busy, available for chat, etc.).

As opposed to email, instant message conversations take place in near-real-time. Individual IMs tend to be quite short – a single question or statement. These brief typed messages are exchanged back and forth between two individuals in a manner similar to the way utterances are exchanged in face-to-face or telephone conversations. You type a message; it pops up on your buddy's desktop. She types a message; her response pops up on your desktop. And so on.

Web browsers, such as Microsoft Internet Explorer, Google Chrome, and Mozilla Firefox enable people to view and navigate web pages over the Internet. Web pages are similar to the pages of a magazine in that they consist primarily of text and color graphics. But unlike magazine pages, web pages can contain other types of content, such as sound, animations, and video. Another difference is that individual web pages may be linked to other web pages forming a vast network (or web) of such pages. End-users move from one web page to another by clicking the mouse on an appropriate word, phrase, or image that is linked to the target page.

More precisely, a web browser is a computer program that interprets and displays documents written in the HyperText Markup Language (HTML). The web (technically called the World Wide Web) is the vast collection of interconnected HTML documents, or web pages, that are delivered via the computers and networks of the Internet.

Web pages are HTML documents that generally combine three technologies: multimedia, hypertext, and programming logic (in the form of JavaScript, a computer programming language that is understood by most web browsers). Multimedia refers to the ability to include multiple kinds of media such as text, graphics, photos, and audio, in a single document. Hypertext is the ability to jump from document to document by clicking on highlighted objects (e.g., words or pictures) that appear within the page.

JavaScript code can be embedded directly into web pages giving them limited program-based functionality. JavaScript tends to be used to do simple things such as presenting an input form to a user, and then checking data as it is entered by the user to ensure that it is of the appropriate type and within expected ranges.

In addition to being used for recreational and general informational purposes, web browsers have become important productivity tools in the workplace. One reason for this is that the underlying document formatting language used by the web, HTML, is a platform neutral way of representing information. Platform neutral means that an HTML document will look pretty much the same regardless of whether it is viewed using Microsoft Internet Explorer, Google Chrome, or Mozilla Firefox, and regardless of whether the browser is run on a Mac, a PC, or a Unix workstation. HTML is important because it allows companies to create documents once and be assured that they can be viewed by employees regardless of the type of computer they have on their desk or which browser they happen to be using.

For vast majority of the 20th century the three essential pieces of office equipment were the telephone, the typewriter, and the filing cabinet. The last decade of the 20th century saw this list change to the telephone, the computer, and the laser printer. Typewriters have now all but vanished from most offices, replaced by computers running word processors and inexpensive high-quality laser printers.

A **word processor**, such as Microsoft Word or LibreOffice, is a computer program that can be used for typing and editing documents. In addition to supporting basic text entry, most word processors include spell checking and grammar checking tools, numerous formatting options, fonts, and page layout features, plus the ability to include images and create simple drawings and graphics.

Word processors were able to so quickly replace typewriters because of their many advantages. Typewriters were designed for one purpose: typing documents. Word processors, on the other hand, are usually just programs running on general-purpose personal computers. This arrangement is much more flexible since the computer can be used for many things other than typing, like accessing a database or performing spreadsheet calculations.

Word processors also make the process of correcting typographical errors much easier than with a typewriter. On a typewriter, once a key had been struck, the character was actually printed on paper. While it was reasonably easy to correct simple mistakes on a good quality typewriter, even those often left some traces of the original error. More substantial corrections, such as inserting a missing word, sentence, or paragraph, were impractical, even on the best typewriters.

In addition to the increased flexibility and improved ability to correct mistakes, word processors allow documents to be automatically formatted in various styles, such as a memo, letter, or report. Word processors can also automatically generate page numbers, place footnotes at the bottom of a page and number them, create indexes, tables of content, and lists of figures; plus a host of similar operations that make creating large documents, such as a book, much easier to do.

Another important feature of word processors are their ability to perform spell checking. Spell checking is possible because a dictionary is included with the word processing program. When a document is spell checked the computer compares every word in the document against the entries in its dictionary. Words that appear in the document but not in the dictionary are flagged as possibly incorrect.

Up until the past few years, scientific, technical, and even most business presentations used very few visual aids. The three immutable features of professional presentations the world over were: the speaker, the overhead projector, and a stack of transparencies. The transparencies were usually simple black and white outlines of the talk, predominately consisting of text with few graphics or images. In this simpler time, the amount of effort the speaker had put into preparing the talk could usually be discerned by noting whether the transparencies were hand written or had been prepared on a word processor.

These days, such presentations are gone. Professional presentations now routinely include an audiovisual slideshow, generated by a laptop computer connected to a portable display device such as a projector. The computer programs used to create and deliver the audiovisual aids for such presentations are known as presentation software, or presentation graphics programs. One of the most popular programs of this type is Microsoft PowerPoint.

Presentation software allows end-users to create professional looking presentations that include text, graphics, high-resolution color photographs, sound effects, and music; together with narration and even limited amounts of video. In addition, visual effects such as “fades” between slides, panning and zooming, and rotation of images and text are usually supported.

Presentation software packages have become an indispensable communication tool redefining the “look” of professional presentations; however, like most tools they must be properly used to increase

productivity. Two common mistakes that can negatively affect productivity are: (1) devoting an excessive amount of time to creating a presentation; and (2) loading a presentation down with attractive “eye candy” that distracts the audience and hinders their ability to concentrate on the point of the presentation.

As presentation packages have become more and more sophisticated, in many ways creating professional looking presentations has become easier, thus saving time and increasing productivity. However, it is also the case that, as more and more capabilities have been added to presentation software, there has been an increasing temptation on the part of end-users to construct more and more elaborate presentations (e.g., adding animations, narration, video clips, etc). As the bar of what is considered an acceptable presentation is raised, many professionals find themselves devoting increasing amounts of time to generating more and more elaborate presentations.

Spreadsheets

Spreadsheet programs, such as Microsoft Excel and LibreOffice Calc, are general-purpose modeling tools that allow end-users to represent a problem as a collection of data items and the formulas that express the relationships between those items. Spreadsheets are used extensively in business, science, and engineering to visualize data (e.g., display sales figures in bar charts and pie charts), to analyze data (e.g., compute sums and averages), and to help answer “what if” type questions (e.g., “What will happen to profitability if wages are increased by 10%?” or “What will happen to payload capacity if overall vehicle mass can be reduced by 5%?”).

Many people consider spreadsheets to be the single most important application that led to the widespread adoption of PCs by businesses in the early 1980s. Dan Bricklin and Bob Frankston developed the first spreadsheet program, VisiCalc, in 1978. Prior to its development, data analysis was either done by hand (i.e., using pencil, paper, and a calculator) or required the use of special-purpose computer programs. These programs had to be designed, written, debugged, and tested – a time consuming and expensive process. With a spreadsheet, budgets and financial forecasts that took weeks to prepare the old fashioned way could be created in a few hours. This dramatic rise in productivity easily justified the cost of a computer and software.

A spreadsheet consists of a rectangular grid, or table, of “cells.” The table is divided into horizontal rows (usually numbered 1, 2, 3, etc) and vertical columns (usually labeled A, B, C, etc). The location of each cell in the table can be specified by its column letter and row number. For example, the upper-left-most cell of a spreadsheet is labeled A1, since its location is column A, row 1. Cell locations are always specified with the column designator listed first, followed by the row designator.

A small spreadsheet containing four columns and eight rows is shown below. Notice that the word “Grades” is stored in cell B2. The value “100” is stored in cell C3. In order to use a spreadsheet program effectively, the first task you must master is to become comfortable with the cell reference system.

	A	B	C	D
1				
2		Grades	50	
3			100	
4			75	
5				
6		Average	75	
7				

Each cell of a spreadsheet may hold one of three things: a number, a text string, or a formula. Numbers are generally composed of the digits 0 through 9 together with an optional decimal point and sign. Hence, “50”, “50.0”, “+50”, and “+50.0” are all valid ways of representing the number fifty.

The standard way that values are entered into a spreadsheet cell is to click the mouse on that cell and then type in the value to be inserted. For example, the value of “50” could have been placed into cell C2 of the spreadsheet above by first clicking on the cell and then typing “50” followed by the “enter” key.

Even though spreadsheets are primarily used to manipulate numbers, there is often a need to include text in the table for such things as titles and column headings. In general, text strings may consist of any sequence of characters. As an example, consider the cell B2 in the spreadsheet above, which contains the character string “Grades”. Similarly, cell B6 holds the text string “Average.”

The final type of item that can be stored in a spreadsheet cell is a formula. Formulas are used to represent mathematical expressions that are to be calculated. As a result, they look and behave very similar to standard mathematical expressions. At a minimum, every spreadsheet program will support the four basic mathematical operators: addition, denoted by “+”; subtraction, denoted by “-”; multiplication, usually denoted by “*”; and division, denoted by “/”.

In addition to the mathematical operators, formulas may contain the numbers upon which the operators are to act. Hence, “=5*3” and “=1+2+3” are both valid spreadsheet formulas. In general, formulas are evaluated from left to right with all multiplication and division being performed before any addition or subtraction. So, “=1+2+3” is computed as 1+2, giving 3, followed by 3+3, giving 6.

Just as with mathematical expressions, parenthesis may be used to override this default ordering and force operations to be performed in any order you desire. So, “=1+(2+3)” would be evaluated as 2+3, giving 5, followed by 1+5, giving 6.

While spreadsheet formulas and mathematical expressions are quite similar, one difference between the two is that spreadsheet formulas use cell references, rather than variables like X and Y. There are other differences as well. In order to begin to understand these differences, let’s consider how we would go about constructing a spreadsheet that would convert a temperature expressed in degrees Celsius into one

expressed in degrees Fahrenheit. The standard equation that expresses the relationship between these two temperature scales is:

$$F = \frac{9}{5}C + 32$$

This expression cannot be entered directly into a spreadsheet for two main reasons: first, it refers to the variables F and C rather than to spreadsheet cell locations; and second, the expression is an equation. The first point is rather obvious; the second point is subtle but important. Mathematical equations are essentially statements of fact, which say that two quantities are equal. Spreadsheet formulas, instead of expressing equivalence, specify a sequence of mathematical operations that are to be performed.

In order to construct a spreadsheet to perform temperature conversions from Celsius to Fahrenheit, we must first decide which spreadsheet cell will contain the input (i.e., the temperature in degrees Celsius) and which cell will hold the output (i.e., the same temperature expressed in degrees Fahrenheit). Let's say that we decide to place the input in cell B2 and would like the result to be placed immediately below it, in cell B3.

Under these conditions, B2 will take the place of C in the above expression. But, what about F? It will not appear directly in the formula. Instead, B3, the cell corresponding to F, will be the cell in which we place the formula that describes how to perform the temperature conversion. Specifically, the following formula will be placed in cell B3: `=9/5*B2+32`

This formula instructs the spreadsheet to divide 9 by 5 and then multiply the result by whatever value is stored in cell B2, and finally to take that result and add 32. The final result of this computation will then be displayed in the cell that contains the formula, cell B3 in this case. Note that the symbol for multiplication must be included between 9/5 and B2. Even though the rules of algebra allow multiplication to be implied by writing two quantities next to one another, spreadsheets require that all multiplication operations be explicitly indicated by the "*" symbol. Many people familiar with spreadsheet formulas would also add parentheses around 9/5, giving `=(9/5)*B2+32`, to make the formula easier to read, even though the parentheses are not strictly required in this case.

The above example is illustrated below. The number 0 is stored in cell B2, in order to represent an input of zero degrees Celsius. Cell B3 holds the formula that will compute the equivalent temperature in degrees Fahrenheit. In order to improve the readability of the spreadsheet, the text strings "Celsius" and "Fahrenheit" have been placed in cells C2 and C3, respectively. Note that this example includes all three types of data that a spreadsheet may hold: numbers, formulas, and text strings.

	A	B	C	D
1				
2		0	Celsius	
3		32	Fahrenheit	
4				
5				

It is important to clearly understand that cell B3 in the spreadsheet above contains a formula and not a number. The value 32 is displayed in cell B3 because that is the value currently being computed by the formula stored in the cell. In order to see the formula underlying a cell, simply click on the cell. In the spreadsheet above, cell B3 has been highlighted and its underlying formula is displayed immediately above the spreadsheet.

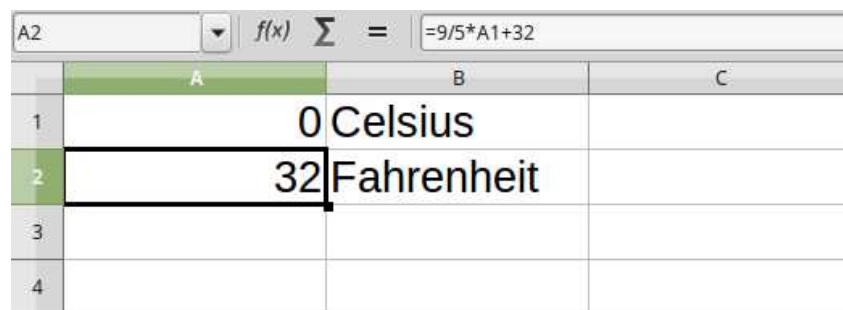
At this point it would be natural to think to yourself, “Wow, that was a lot of work to go through just to show that 0 degrees Celsius is equal to 32 degrees Fahrenheit.” And you would be right if that was all this spreadsheet could do.

The power of spreadsheets lie in their ability to automatically recalculate and update values that depend on other values. In the case of the current example, changing the value stored in cell B2 will cause the formula in cell B3 to be recomputed. So, if the value of 100 were placed in B2, the value displayed in B3 would automatically change to 212. The Fahrenheit equivalent of any temperature expressed in degrees Celsius can be computed just as easily: simply enter the temperature in Cell B2.

Relative and absolute cell referencing

A feature that adds to the flexibility of spreadsheets is their ability to move or copy the contents of cells from one location in a table to another. This feature is best explained by reference to an example.

Recall the spreadsheet for converting temperatures from Celsius to Fahrenheit that was introduced earlier. Suppose that we wish to move the contents of the cells by deleting the first row and first column. This would effectively shift the cell contents up one row and left one column as follows:



	A	B	C
1	0 Celsius		
2	32 Fahrenheit		
3			
4			

Comparing the spreadsheets proves that the “move” operation (or the deletion of the row and column) does exactly what we would expect it to do. It moves the spreadsheet for temperature conversions to a different portion of the table. However, a careful comparison of the spreadsheets reveals that “move” involves more than at first meets the eye. When the formula $= (9/5) * B2 + 32$ stored in cell B3 was moved to cell A2, it was modified to read $= (9/5) * A1 + 32$. While this change might at first seem perplexing, it is necessary in order to allow the temperature conversion spreadsheet to work properly at its new location in the table. If the formula had not been modified, it would be in error after the move since cell B2 no longer holds the temperature to be converted.

When performing a move or copy operation, numbers and text strings are copied to their new location without change. However, when moving or copying a formula, a technique known as relative cell referencing is usually used. **Relative cell referencing** treats the references to cells in a formula as being relative to the current location of that formula. For example, if a formula located in cell A2 makes reference to cell A1, that reference is interpreted to mean the cell immediately above the current cell. It is this relative location that is preserved in a move or copy operation. If the formula were moved to

location B3 then the cell reference would be changed to B2, so that after the move the formula still refers to the cell immediately above it.

This concept of relative cell referencing is further illustrated below, which shows how a formula that is located in cell B3 would be modified if copied to C7. In its original location, the formula appears as =A1+C2. Since cell A1 is located two cells up and one cell to the left of cell B3, when the formula is moved or copied to C7 this reference is changed to B5. Note that B5 is exactly two cells up and one cell to the left of C7. Likewise, the reference to cell C2 in the original formula, which is one cell up and one cell to the right of B3, will be changed to D6 when the formula is moved to C7 since D6 is one cell up and one cell to the right of C7.

	A	B	C	D
1				
2				
3		=A1+C2		
4				
5				
6				
7			=B5+D6	
8				

As we have seen, relative referencing gives us the flexibility to freely move a spreadsheet to any location in the table. Most of the time, relative cell referencing is what we want. However, there are occasions when it is useful to be able to “lock” a cell reference in a formula to a particular location. This is called **absolute cell referencing**. In absolute cell referencing, even if the formula is moved, the referenced cell will not change.

Absolute cell referencing comes in most handy when there are one or more values that will be referenced in many different formulas. For example, say you were constructing a financial spreadsheet that dealt with employee salaries. It is likely that the current federal, state, and local income tax rates would be used in many calculations throughout the entire spreadsheet. Placing these rates in fixed cells and having all formulas that use these rates refer absolutely to those cells would be a very good idea. This is because tax rates inevitably change over time. When a tax rate increases, the only change that will have to be made to the spreadsheet will be to insert the new rate in the appropriate cell. If you had instead chosen to treat the tax rates as constants and embed them directly in the underlying formulas, then whenever a rate changed you would have to find and change every formula in the spreadsheet that used that rate.

Constructing formulas that refer to the tax rates using relative cell referencing would be better than using constants, but still not as good as absolute referencing. The reason for this is that many formulas are likely to use the tax rate information. If you used relative referencing, then you would not have the flexibility to move the formulas around in the table. Every time you tried to move a formula that referred to a tax rate, relative referencing would corrupt your formula.

One common way of indicating that cell references are to be absolute is to place a dollar sign symbol in front of both the column and row designators. For example, if we wanted the reference to cell A1 to be

absolute instead of relative, we would write $\$A\1 . The spreadsheet below illustrates the effect of moving a formula that contains both absolute and relative references. The original formula, located in cell B3, contains an absolute reference to cell A1 and a relative reference to cell C2. After a move to cell C7, the absolute reference to cell A1 remains unchanged, while the relative reference is modified to point to D6.

	A	B	C	D
1				
2				
3		$=\$A\$1+C2$		
4				
5				
6				
7			$=\$A\$1+D6$	
8				

One final point about cell referencing. As you know, a cell reference consists of two parts: the column reference and the row reference. It is possible to refer to one of these absolutely and the other relatively. For example, $\$A1$ “locks” the column reference but allows the row reference to remain relative. $C\$2$ works in the opposite manner, allowing the column reference to be relative while forcing the row to remain fixed.

The spreadsheet below illustrates the effect of “mixed” cell references. As in the previous two spreadsheets, the formula originally located in cell B3 refers to cell A1 and C2, and is moved to cell C7. The reference to A1 is $\$A1$, which locks the column to “A”, but allows the row reference to remain relative. Since row 1 is two rows above row 3, when the formula is moved to row 7 the referenced cell will be on row 5 (two rows above row 7). Hence the complete reference will be $\$A5$. The reference $C\$2$ is handled in a similar manner, except the row is locked to 2, while the column is relative (one column to the right of the current cell).

	A	B	C	D
1				
2				
3		$=\$A1+C\2		
4				
5				
6				
7			$=\$A5+D\2	
8				
9				

In the preceding discussion, the terms “move” and “copy” were used somewhat interchangeably. This is because both operations handle cell contents in the same way. The only difference between the two is

that “move” erases the contents of the source cells after performing a copy operation. Hence, after a “copy” there will be two instances of cell contents: the original (source) contents, and the copied (destination) contents. After a “move” there will be only one instance of the cell contents, in the destination location.

Replication and built-in functions

When working with spreadsheets it is frequently the case that a column (or row) of the spreadsheet will contain a large number of very similar formulas. For example, the spreadsheet below is used to calculate the effect of a projected 10% raise on employee’s salaries. Column A contains the names of the employees, column B contains the salaries for the current year (which is given as 2215), and column C contains the salaries for the following year (2216). An examination of column C reveals that all of the formulas in cells C2 through C6 have the same general form: multiply the contents of the cell immediately to the left of the current cell by 1.1. These formulas have the effect of computing a 10% raise. What is the best way of entering these formulas into the spreadsheet?

	A	B	C
1	Name	2215	=B1+1
2	April, R.	50000.00	=B2*1.1
3	Pike, C.	55432.10	=B3*1.1
4	Kirk, J.T.	69666.42	=B4*1.1
5	Picard, J.	72123.45	=B5*1.1
6	Janeway, K.	888888.00	=B6*1.1

The resulting spreadsheet is shown below:

	A	B	C
1	Name	2215	2216
2	April, R.	50000.00	55000.00
3	Pike, C.	55432.10	60975.31
4	Kirk, J.T.	69666.42	76633.06
5	Picard, J.	72123.45	79335.80
6	Janeway, K.	888888.00	977776.80

The most straightforward way of entering the formulas is to type each one by hand. While this method is obvious, it is also time consuming. A far better approach would be to use the “copy” and “paste” features built into most spreadsheet programs. Interestingly, spreadsheet programs generally allow you to copy the contents of a single cell to the “clipboard” and then paste the contents of that cell into an entire block of cells simultaneously. In addition, if the content of the cell being pasted is a formula, that formula’s relative cell references will be respected. This combination of features often turns out to be very useful.

For example, to produce the raises shown in column C of the spreadsheet above, a sequence of actions similar to the following could be executed: type the original formula =B1+1.1 into cell C2, then click on the cell, and select the “copy” operation (or press Ctrl+C); next, highlight the block of cells beginning at C3 and ending at C6 (i.e., by clicking on cell C3 and dragging the mouse to cell C6); finally, click “paste” (or press Ctrl+V). All of the cells from C3 to C6, inclusive, will be filled with copies of the formula in cell C2 (modified in the standard way).

Another feature common to most spreadsheets is built-in functions. Built-in functions provide a convenient way of expressing frequently used computations, especially those that involve references to large blocks of cells. For example, most spreadsheet programs include functions for computing the “sum” and “average” of any block of cells. Without functions, computing the average of the values in cells C1 through C10 would require the user to enter a formula such as the following:

$$=(C1+C2+C3+C4+C5+C6+C7+C8+C9+C10)/10$$

Imagine how long the formula would be if we needed to average the contents of 50 cells values rather than just ten. Besides requiring a lot of typing, such an approach is prone to error. For example, the user must make sure s/he correctly counts the number of items to be averaged. If the sum is divided by the wrong number of items, the computed result will be incorrect.

A common way of expressing functions is to list the function name followed by an indication of the block of cells the function will act on, usually listed in parenthesis. For example the function to average the contents of cells C1 through C10 might take the form =AVERAGE(C1:C10)

In this case, AVERAGE is the name of the average function, and C1:C10 indicates that C1 is the first cell in the row or column to be averaged and C10 is the final cell. While this formula could be typed directly into a cell, it can also be created by clicking the series of menu items to insert a function. In fact, doing it this way will show you the variety of spreadsheet functions available to you (there are many!).

Solving problems with spreadsheets

Now that we have a good idea of the nuts and bolts that make up spreadsheet programs, it is time to turn our attention to the question of how spreadsheets can be used to solve “real-world” problems. As mentioned above, one of the primary uses of spreadsheets is to answer “what if” type questions. One such question that most students are interested in is “What will be my grade be if I make X on the final exam?” Let’s construct a spreadsheet to help answer this question.

The first thing we need to know in order to create a spreadsheet is what the input data will be, what outputs are expected, and what mathematical relationships will exist between the inputs and the outputs. For the problem of computing a course average, we know, in general, that the inputs will be exam and homework grades and that the output will be the final average. The relationships that exist between these inputs and outputs are usually specified by the instructor’s grading policy. Here is a grading policy similar that seems pretty straightforward:

There will be three exams given in this course. Exams one and two will each be worth 20% of your final grade. Exam three will be worth 30%. Homework will be worth a total of 30% of your final grade. Four homework assignments will be given. All exams are graded on the 100 point scale. Homeworks will be graded on a 20 point scale.

From this grading policy, we can see that there will be exactly four homework grades and three exams. To begin constructing the spreadsheet, let's have one column for homework grades and one column for exam grades. If we select column B for the homework and column D for exams, we might want to label the columns; say, by placing the text string "Homeworks" in cell B1 and "Exams" in cell D1. We can now place our homework scores in cells B3, B4, B5, and B6. Exam grades could be stored in cells D3, D4, and D5. The spreadsheet (with perfect scores for each homework and exam) below illustrates (along with calculations that will be discussed below):

	A	B	C	D	E
1		Homeworks		Exams	
2					
3		20		100	
4		20		100	
5		20		100	
6		20			
7					
8	Averages	20 / 20		100 / 100	
9	Points earned	30		70	
10					
11	Final score	100			

Note that additional titles have been added to improve readability (i.e., "Averages" in cell A8, "Points earned" in cell A9, and "Final score" in cell A11). Scale indicators were also added in cells C8 and E8. These are text strings of the form: "/20" and "/100".

Remember that there is nothing special about this particular arrangement of the data. One could simply have thought that grouping the grades into columns would make the information easier to read, and that blank spaces between rows and columns may improve readability.

Now that we have decided where the inputs should go, we can begin to think about computing the averages. Instead of trying to come up with a complex formula that directly computes the course average, let's address the task in stages, first computing a homework average, then an exam average, and finally a course average.

The homework average can be computed either by using the built-in "average" function or by directly specifying the formula $=(B3+B4+B5+B6)/4$. The exam average cannot be computed in such a straightforward manner due to the fact that all exams are not weighted equally. Exams 1 and 2 are each 20% of the final grade, while exam 3 is 30%. Hence, all three exams together make up only 70% of the overall grade. These relationships can be captured by the following formula:

$$=(0.2*D3+0.2*D4+0.3*D5)/0.7$$

This formula expresses the exam average using the 100 point scale. Each of the exam grades is multiplied by its appropriate weight and summed, giving the number of points earned from exams. This value is then divided by 0.7 since the total number of points possible on exams is 70% of the total possible points. For convenience sake, the formula for computing the homework average has been placed in cell B8, and the formula for computing the exam average has been placed in cell D8.

Now that the homework and exam averages have been computed, we can work on combining them to produce the overall class average. Unfortunately, we cannot simply add the two averages together and divide by two. The homework average is expressed on a 20-point scale and is worth 30% of the final grade. The exam average is expressed on the 100-point scale and is worth 70% of the final grade.

One approach is to next determine how many of the total points of the final grade will be contributed by homework and how many by exams. The number of points contributed by the homework average is given by the formula $=B8/20*30$, since $B8/20$ is the percentage of homework points earned, and 30 is the total number of points homework is worth (on the 100 point scale). Similarly, the number of points contributed by the exam average is given by the formula $=D8/100*70$, since $D8/100$ is the percentage of exam points earned, and 70 is the total number of points exams are worth (again, on the 100 point scale). Storing the points earned directly under the averages can be accomplished by placing the formula for homework points in cell B9 and the formula for exam points in cell D9. Finally, the course average can be computed by adding together the points earned on homework with the points earned on exams. This is given by the formula $=B9+D9$.

The spreadsheet below shows the completed spreadsheet with formulas shown:

	A	B	C	D	E
1		Homeworks		Exams	
2					
3		20		100	
4		20		100	
5		20		100	
6		20			
7					
8	Averages	$=AVERAGE(B3:B6) / 20$		$=(0.2*D3+0.2*D4+0.3*D5)/0.7 / 100$	
9	Points earned	$=B8/20*30$		$=D8/100*70$	
10					
11	Final score	$=B9+D9$			

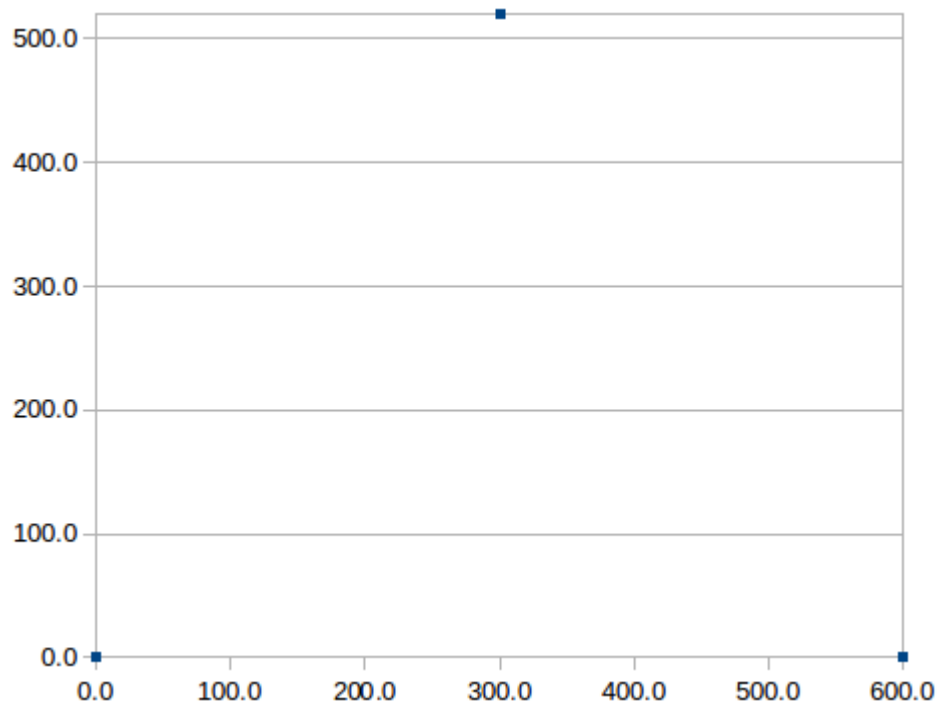
Now that the spreadsheet formulas have been set up, it's easy to play the "What will my grade be if..." game. Simply enter the grades you have made on exams and homeworks that have already been returned and "guesstimate" your grades on the remaining exams and homeworks. Here's an example:

	A	B	C	D	E
1		Homeworks		Exams	
2					
3		19		96	
4		20		87	
5		18		83	
6		16			
7					
8	Averages	18.25 / 20		87.86 / 100	
9	Points earned	27.38		61.50	
10					
11	Final score	88.88			

The chaos game...again?

You've seen (and played) the chaos game several times now. In fact, you've written several programs that generate the Sierpinski triangle by playing the game with three initial points (as the vertices of an equilateral triangle). Spreadsheet programs do more than just allow the tabulation of data and evaluation of formulas. They can also visualize data in a variety of ways. One such way is called a scatter (or xy) plot. From a list of 2D data (or points), a chart can be generated. Here's an example of one, with the spreadsheet first followed by the chart:

	A	B
1	x	y
2	0.0	0.0
3	600.0	0.0
4	300.0	520.0

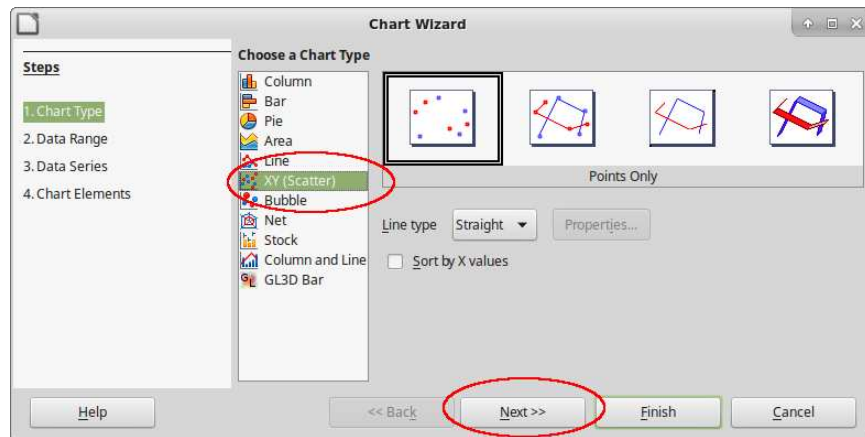


The spreadsheet has headers for the x- and y-component of the points in cells A1 and B1. Although not perfectly equidistant, the points (in cells A1 through B4) are close enough to the vertices of an equilateral triangle. The first point (in cells A2 and B2) is (0.0, 0.0), the second point (in cells A3 and B3) is (600.0, 0.0), and the third point (in cells A4 and B4) is (300.0, 520.0). The chart shows the three vertices plotted exactly as specified in the spreadsheet.

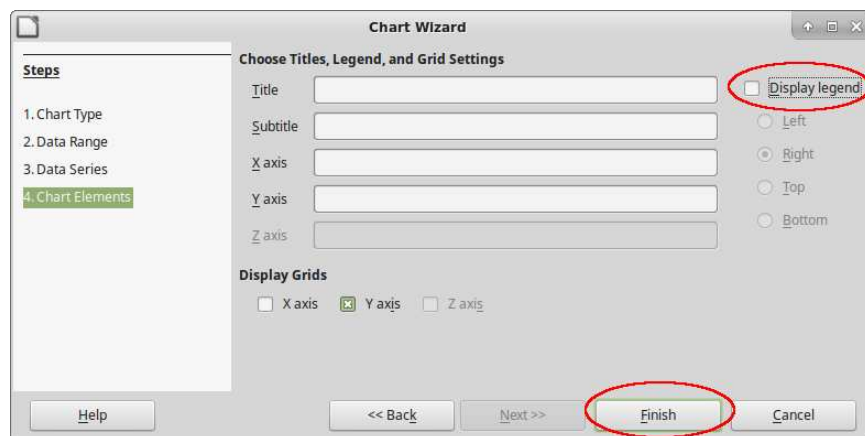
To create the chart, the three points were first highlighted (by clicking on cell A2 and dragging through cell B4). The Insert menu was then clicked, followed by the Chart menu item. Another way of inserting a chart is to click on the Chart button in the toolbar. In LibreOffice Calc, it looks like a pie chart with a slice taken out of it. Note that these methods of inserting a chart work for (and, in fact, may be specific

to) LibreOffice Calc. Although there are other spreadsheet programs available (e.g., Microsoft Excel), this lesson uses LibreOffice Calc to demonstrate the visualization of data.

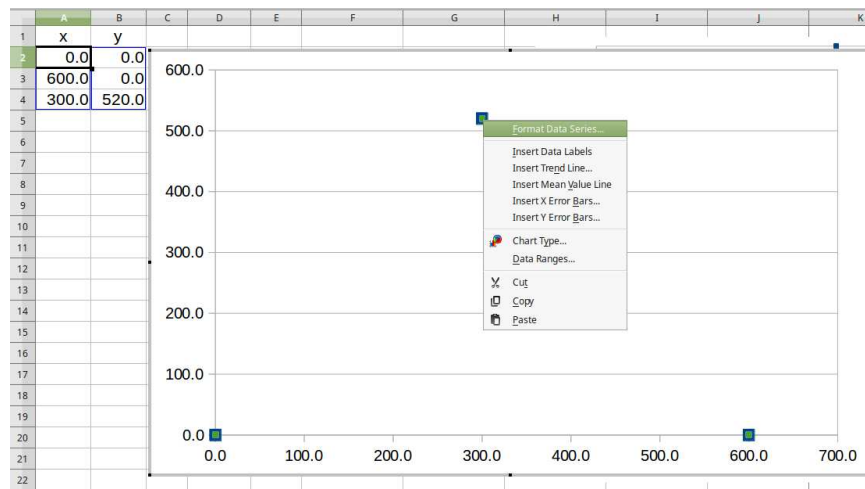
To create the scatter plot show above, insert the chart as described. Select XY (Scatter) as the type of chart, and click Next:



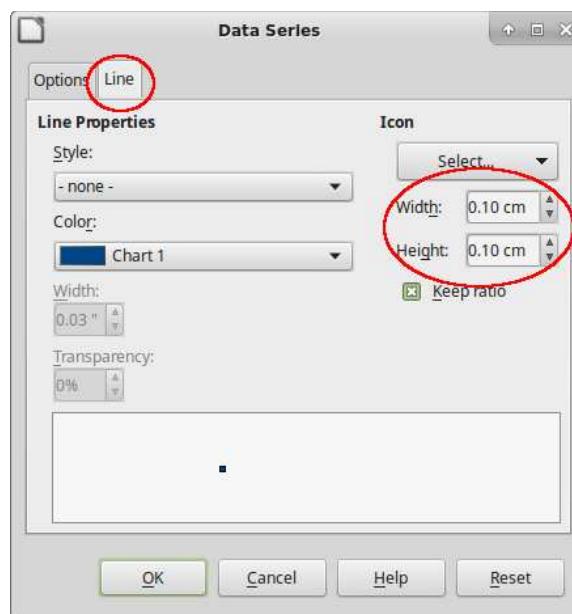
Click Next several times until the last Chart Wizard window with the Display legend checkbox. Uncheck the box and click Finish:



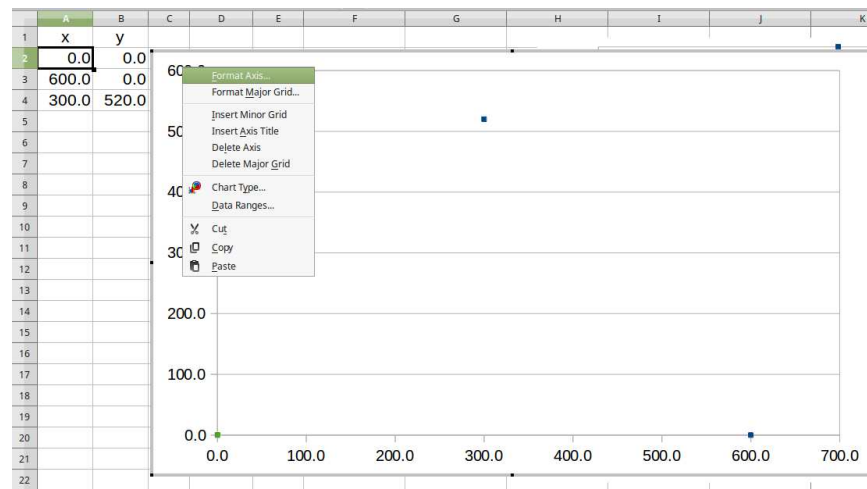
Next, select the chart in the spreadsheet (sometimes, you have to double-click on it). Right-click one of the points and click on Format Data Series:



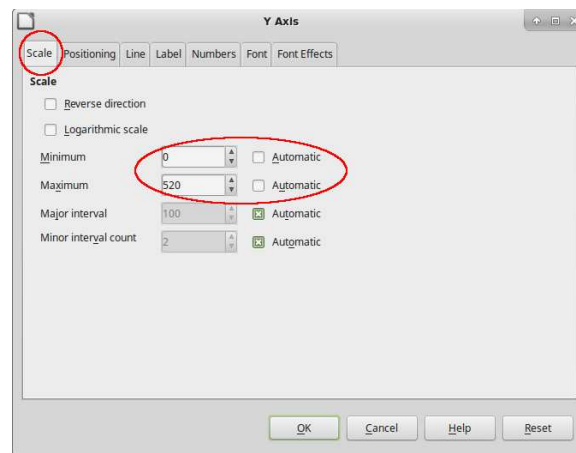
Click the Line tab, and set the width and height of the point icons to be 0.1cm:



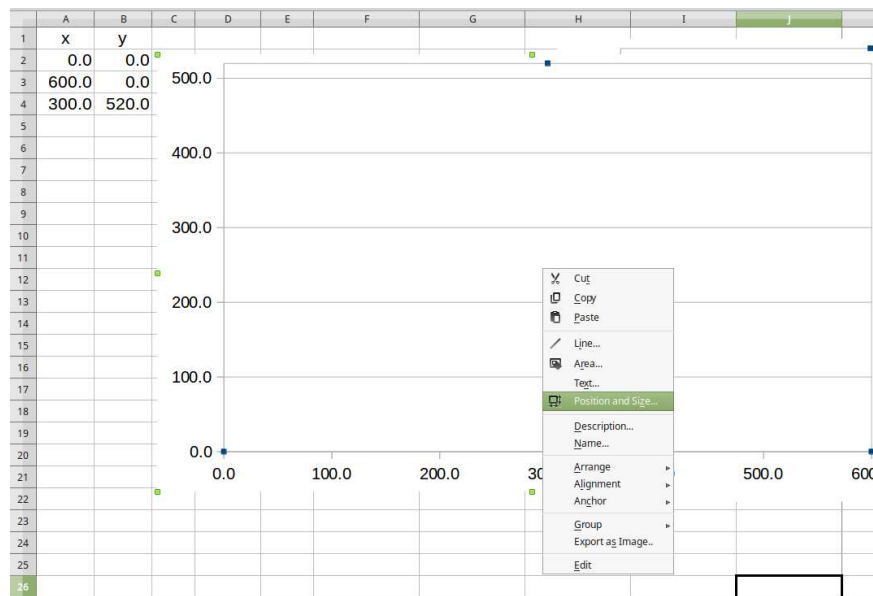
To restrict the x- and y-axis values, right-click any value on the y-axis (e.g., 600.0) and click on Format Axis:



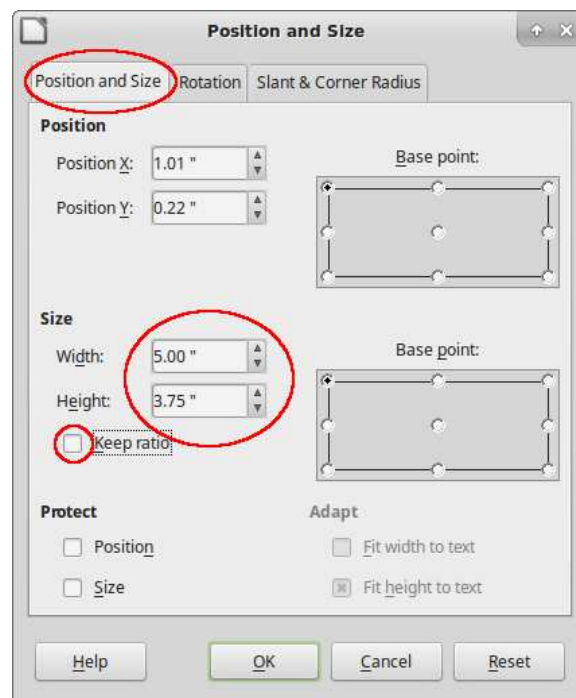
On the Scale tab, set the minimum and maximum to be 0 and 520 respectively (you will need to uncheck the Automatic checkboxes for each):



Lastly, deselect the chart by clicking on a cell in the spreadsheet. Next, right-click the chart and click on Position and Size:



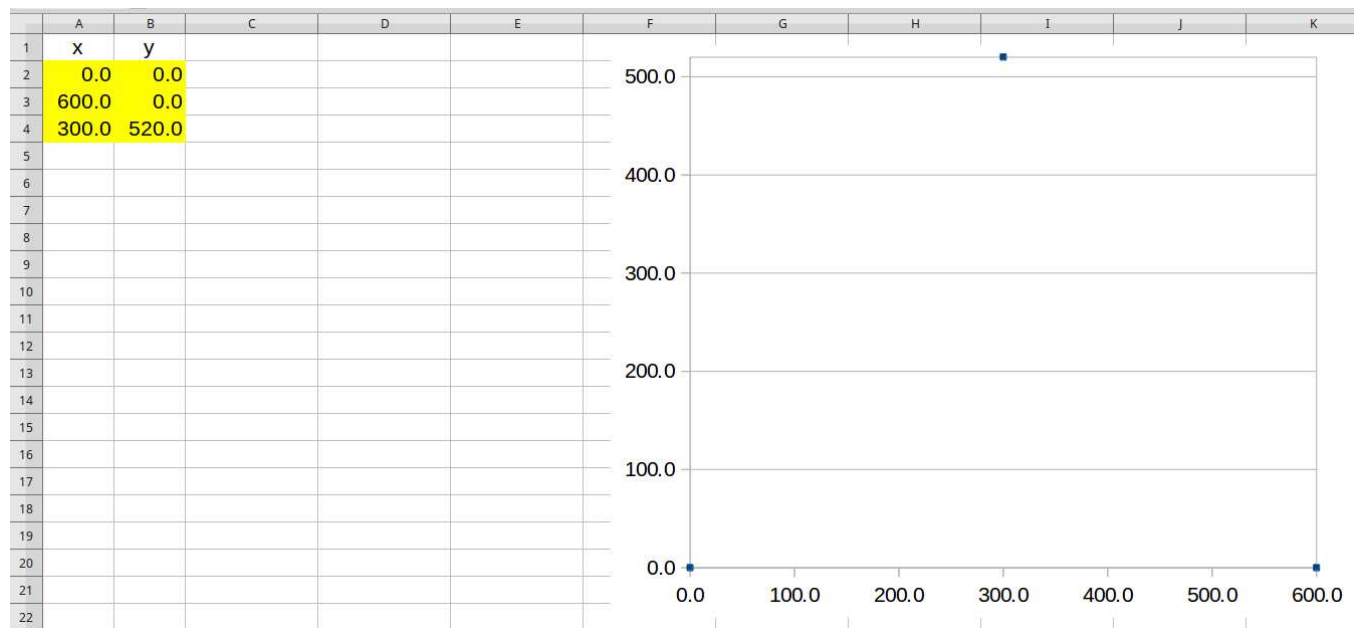
On the Position and Size tab, uncheck the Keep ratio checkbox and set the width and height of the chart to 5.00" and 3.75" respectively:



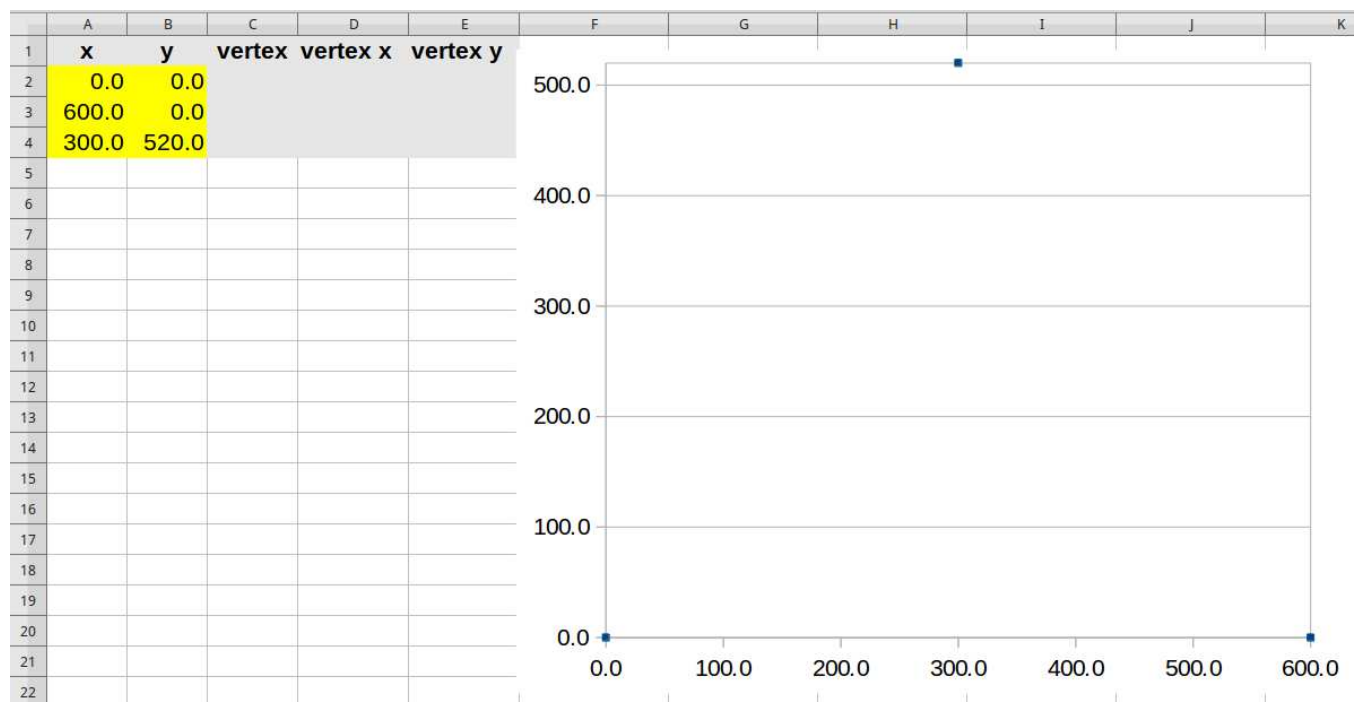
The described procedure should result in a chart of the three vertices that looks exactly as shown earlier.

To play the chaos game and see what happens as more and more points are added (and plotted), we simply need to generate the point data, and then select them all to be plotted on the scatter plot. To

begin, let's highlight the part of the spreadsheet that corresponds to the vertices of the equilateral triangle:



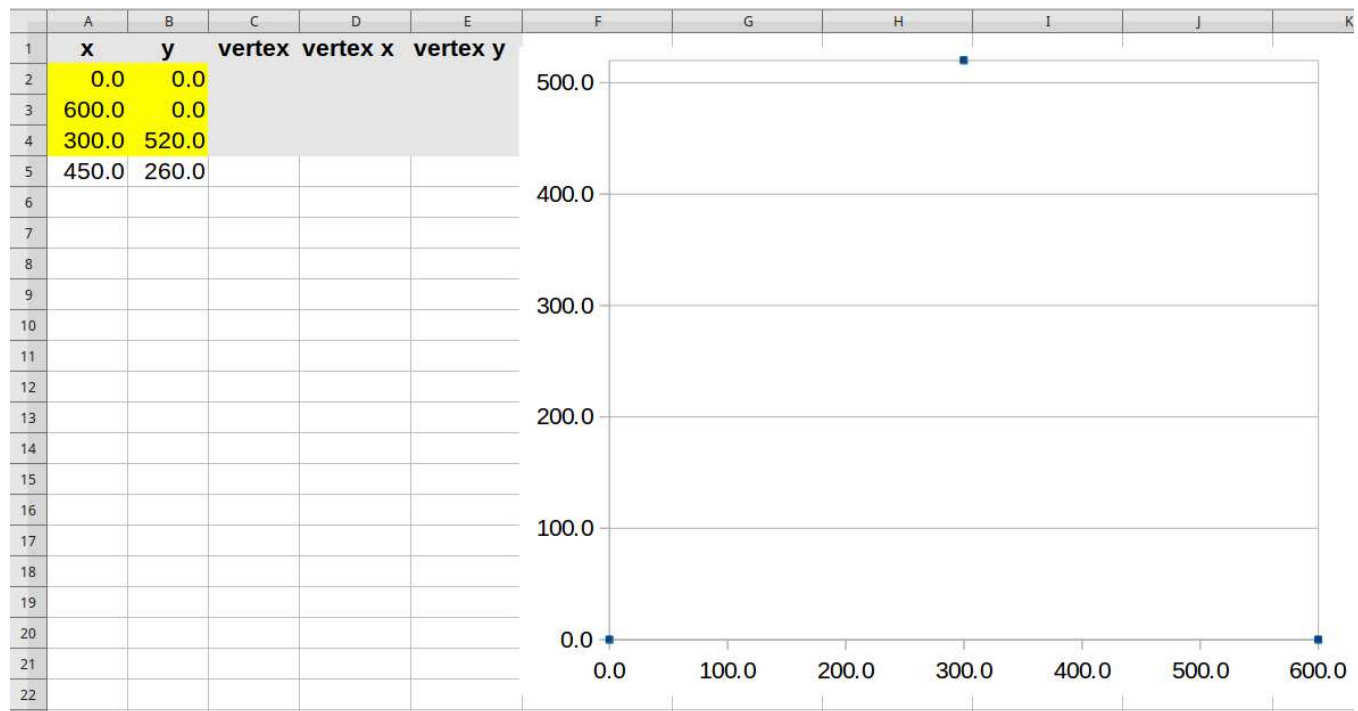
Since we will be randomly selecting a vertex each time, why don't we add columns for the vertex and its corresponding x- and y-components for each of the calculated midpoints that will be generated (note that a few formatting changes have been made for aesthetic reasons):



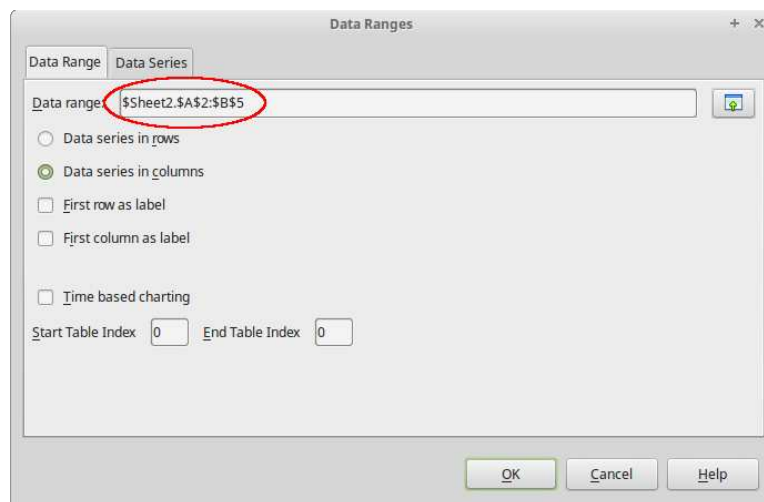
For the first midpoint, we typically randomly select two of the vertices. For simplicity, let's simply select the second (600.0, 0.0) and third (300.0, 520.0). The new midpoint (which will be placed in cells

A5 and B5), will be the midpoint of these two vertices. Thus, we can enter the following formula in cell A5: $=(A3+A4)/2$. Similarly, we can enter the following formula in cell B5: $=(B3+B4)/2$. To represent the x- and y-components with one decimal to the right of the decimal point, we can click on either the Add Decimal Place or Remove Decimal Place icons in the toolbar. Another method of doing this is to select the cells and right-click on them. Then, click on Format Cells, select the Numbers tab, and specify the desired number of decimal places.

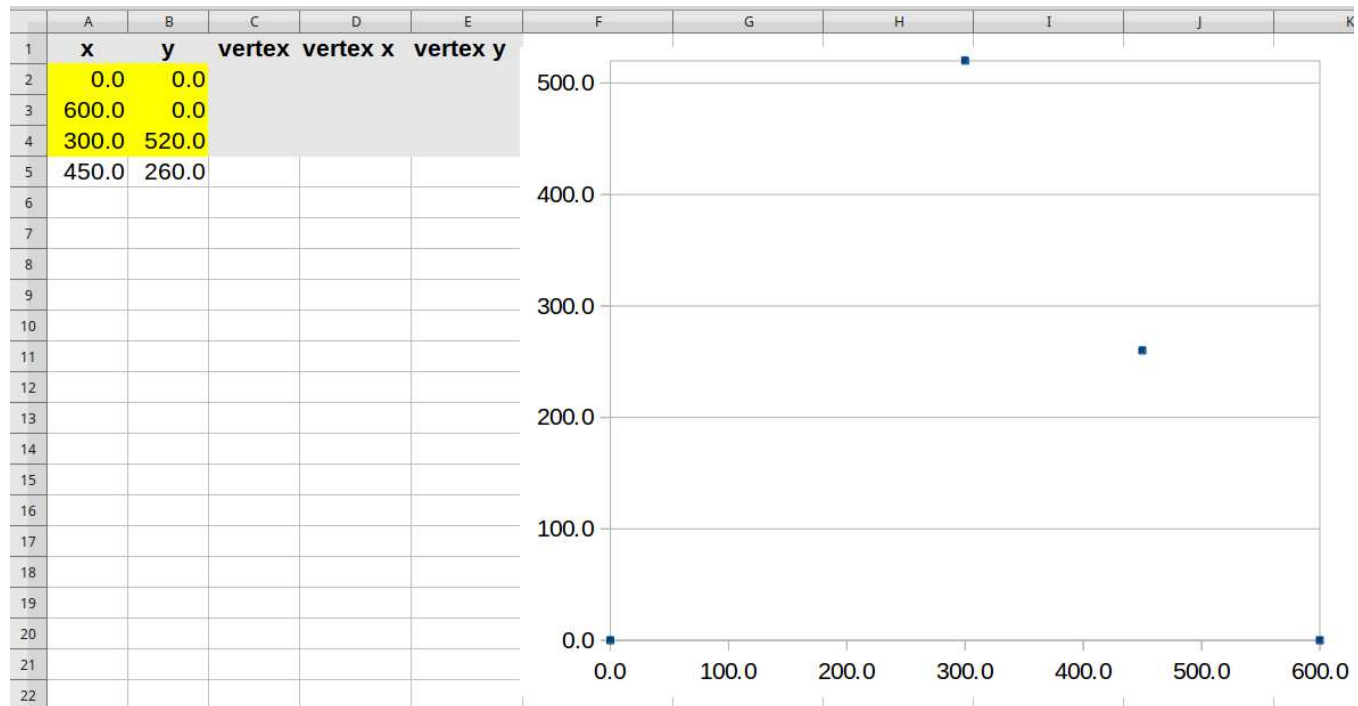
The spreadsheet now looks like this:



Note that the new midpoint has not been added to the chart yet. To do so, we can double-click on the chart, then right-click and select Data Ranges. The data range can be updated from `$Sheet1.$A$2:$B$4` to `$Sheet1.A2:B5`:



Note that the added “\$Sheet1.” means that the data comes from Sheet1 (which should be the current sheet). A single saved spreadsheet can have more than one spreadsheet if desired. Switching between spreadsheets can be at the bottom-left of the interface. The chart should now have the new point added:



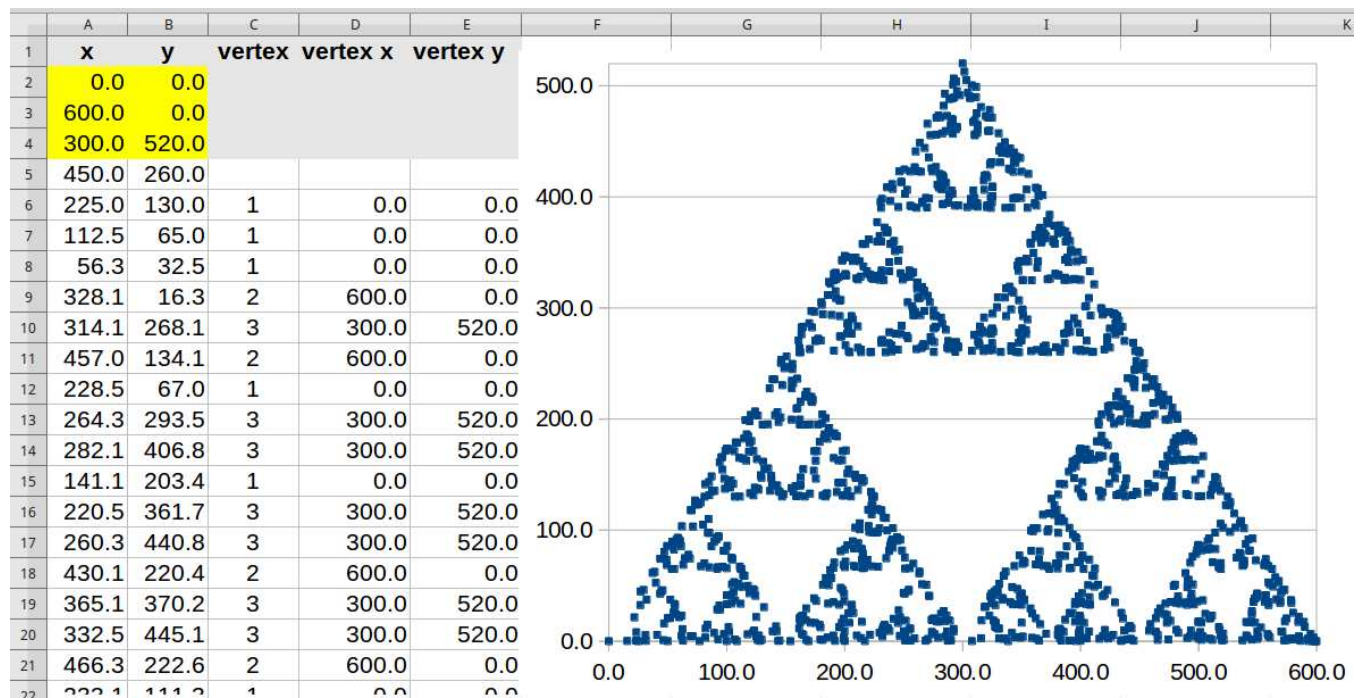
We can now add more midpoints. The first step will be to randomly select a vertex. If we consider the first vertex to be in cells A2 and B2 and the third vertex to be in cells A4 and B4, then we can randomly select one with the following formula: =RANDBETWEEN(1,3). Let's enter this formula in cell C6.

We can then “grab” the x- and y-components of the vertex that corresponds with this randomly chosen vertex and place them in cells D6 and E6. The formula in cell D6 is: =INDEX(\$A\$2:\$B\$4,C6,1). The INDEX function allows the selection of a cell within a group of cells by specifying the range of cells (or reference), the row, and the column. In this case, the range (or reference) is \$A\$2:\$B\$4. Note the absolute cell references. We need these because we will apply this formula for more midpoints, and we do not want the reference cells to shift as we do so. The row within the specified range is the randomly chosen one in cell C6, and the column is 1 (for the x-component). Similarly, we can apply the formula for the y-component in cell E6: =INDEX(\$A\$2:\$B\$4,C6,2). Note the only difference: the column is 2 to specify the y-component of the vertex.

Lastly, we can calculate the midpoint (in cells A6 and B6). The formula in cell A6, which is the x-component of the midpoint of the point immediately above it (in cells A5 and B5) and the randomly chosen vertex (in cells D6 and E6), is: =(A5+D6)/2. Similarly, the formula for the y-component in cell B6 is: =(B5+E6)/2.

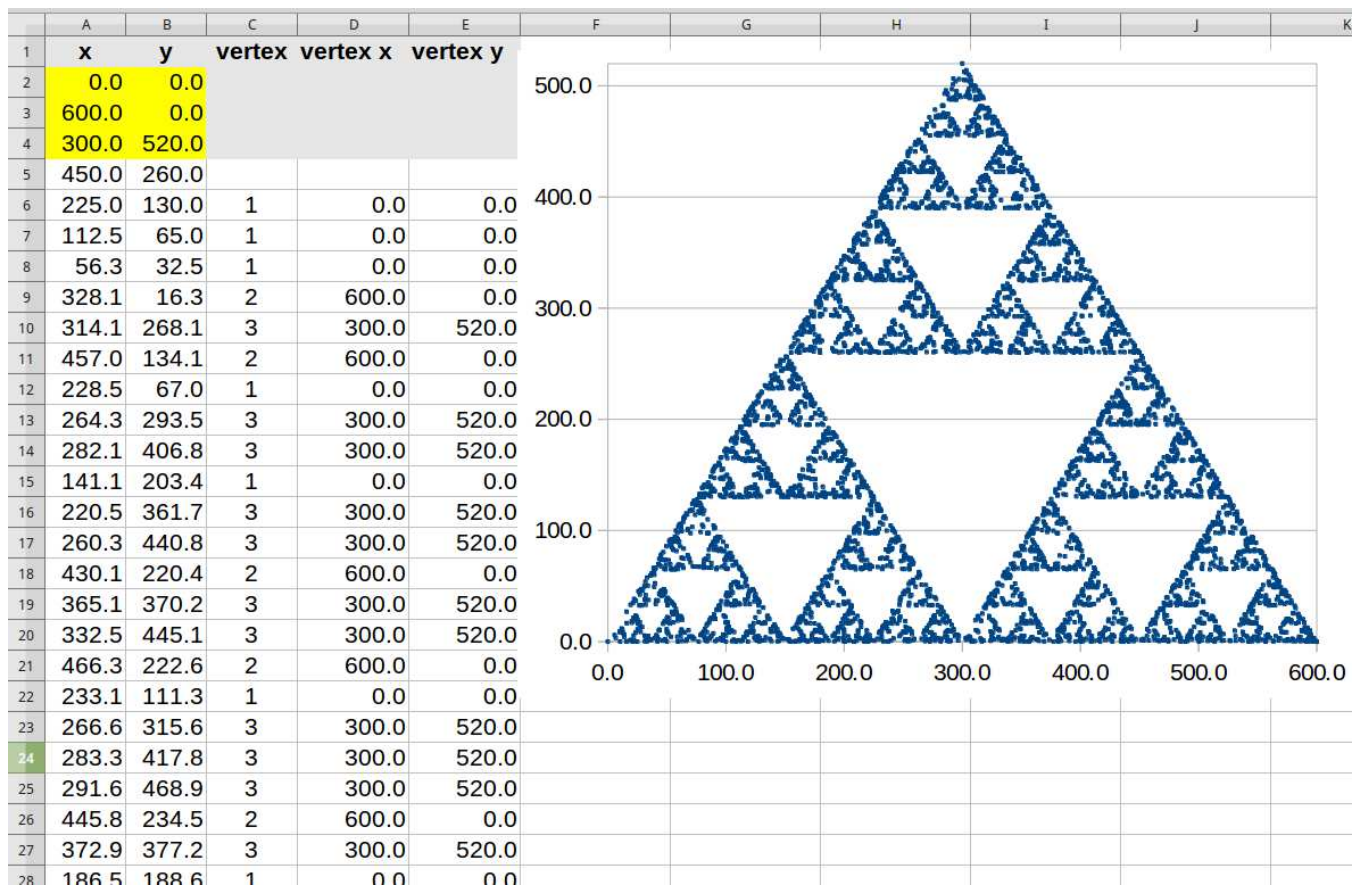
All that's left to do is to apply the formulas of row 6 (in cells A6 through E6) to additional rows beneath row 6! In fact, we can do this by selecting cell A6, scrolling down to, say, row 1500, holding the shift key and clicking on cell E1500, and clicking on Edit, then on Fill, and lastly on Down. This fills the cells (from A7 down through E1500) with the formulas specified in cells A6 through E6!

We can now update the chart by modifying the Data Ranges as before; this time, however, we change the range to \$Sheet1.\$A\$2:\$B\$1500. Here's the resulting spreadsheet and chart:



Note that the entire spreadsheet has not been captured in the figure above (since there are 1500 rows!).

A suggestion may be to reduce the size of the data points on the chart to, say, 0.05cm, and add even more rows (say, to 5000):



Software engineering is the study of the design, construction, and maintenance of large software systems. As the hardware capabilities of computers have increased, so have the expectations for the performance of software. We expect programs to be friendly, easy to use, reliable, well documented, and attractive. Meeting these expectations often increases the size and complexity of a program. Thus, over time, the average size of programs has tended to increase.

Many software systems represent a significant number of person-years of effort and are written by large teams of programmers. These systems are so vast that they are beyond the comprehension of any single individual. This is most likely unlike the kinds of programs that you have worked on (probably by yourself) so far. As computer systems become more and more intertwined into the fabric of modern life, the need for reliable software steadily increases. As an example take the cellular telephone network. This system consists of millions of lines of code written by thousands of people over decades; yet, it is expected to perform with complete reliability 24 hours a day, 7 days a week.

Unfortunately, whereas the scaling up of hardware has been a straightforward engineering exercise, software production cannot be so easily increased to meet escalating demand. This is because software consists of algorithms that are essentially written by hand, one line at a time. As the demands of increasing reliability and usability have led to software systems that can not be understood by a single person, questions concerning the organization of teams of programmers have become critical.

These managerial issues are complicated by the unique nature of software production. Programming is a challenging task in its own right, but its complexity is increased many fold by the need to divide a problem among a large group of workers. How are such projects planned and completion dates specified? How can large projects be organized so that individual software designers and programmers can come and go without endangering the stability of the project? These are just some of the questions addressed by software engineering.

One of the cornerstones of software engineering is the **software life cycle**: a model of how software is developed, used, maintained over time, and eventually discarded. This model is helpful in predicting costs and allocating programming resources, but it suffers from inflexibility. This inflexibility has led to the adoption of alternative software development models in recent years, such as rapid prototyping. Newer models tend to focus on improving user satisfaction with the software (e.g., making programs more usable and user-friendly).

In addition to managing the development of individual software projects, software engineers also design tools for automating portions of the software development process. For example, tools to assist with the creation of program documentation and testing are now common. Such tools are often referred to as CASE (Computer Aided Software Engineering) tools.

Analysis and design

When working on large software projects, rarely do we ever begin in front of a computer monitor and keyboard, and begin to code. For simple programs that you are likely to design during your educational experience, perhaps this is fine. The kinds of projects that computing professionals work on are rarely, if ever, so simple. Just as understanding the problem is crucial to ultimately solving it, so is understanding the requirements of a project in order to ultimately design a solution that does what we

want. A large part of software engineering is analyzing what we want in order to ultimately design what we want.

The process of analyzing what we want in order to specify the requirements of a system can often be time consuming. In fact, this part of the process often involves a customer or client who will ultimately be the recipient of the produced system. You may be thinking that the knowledge base of the customer is probably very different from the knowledge base of a programmer (or an organization that is tasked with designing the system for the customer). And you would probably be right.

The process of analyzing the requirements of a system is often done in a game-like manner, playing out scenarios. One method, called the **verb-noun method**, attempts to identify things or entities involved in the system. These entities are nouns that may ultimately translate to classes and objects in a programming project. In addition, playing out scenarios by having the entities interact with each other helps to discover actions that involve them. These actions, or verbs, may ultimately lead to interactions between objects in a programming project. In fact, this may lead to the discovery and implementation of methods and functions.

One way to help discover entities in a system is to use CRC (Class Responsibility Collaborators) cards. **CRC cards** attempt to capture the classes (nouns), their responsibilities (verbs), and their potential collaborators (other classes that are specified on other CRC cards). The goal of using CRC cards is ultimately to play out scenarios involving the classes. This helps to analyze system requirements and to discover object interactions. In addition, it helps to see if the problem description is clear and complete. Ultimately, class interfaces can be designed from the CRC cards while playing through various scenarios. Below is a sample CRC card:

Class name	
Responsibilities	Collaborators

Documentation

Documentation is arguably very important; yet, it is often the most despised aspect of engineering software. In fact, many developers put it off until the end of a software project. This often leads to inconsistent and incomplete documentation of the project! Usually, it is good practice to write class and

method comments that describe the **purpose** of each. This ensures that the focus of the comments is on *what* instead of *how*. It also ensures that we don't forget what something does or is responsible for.

When working on a software project, it is easy to remember what something does, why decisions were made, and so on. But after being removed from the project for some time, it is often difficult to recall programming decisions. In fact, many programmers often start over on small projects rather than go through existing code that is not commented properly to try to understand it.

Nowadays, there are many tools that can automate the generation of a user manual or documentation of an application using embedded comments in the source code. The use of special, meaningful tags are interpreted by this type of software, which allows it to render the documentation.

Prototyping

Often, we wish to observe and test an application before it is entirely finished. That is, we would like to observe its use in order to perhaps drive further design decisions. Prototyping involves generating a preliminary model of an application to help support early investigation. It allows us to test certain functionality while ignoring other details. Usually, a prototype is not complete; that is, it is not a complete, finished version of the application. Often, the incomplete components are simulated (or sometimes *hard-coded* with temporary values or behaviors that will be replaced later on). In prototypes, we try to avoid random behavior, as it is difficult to reproduce.

Iterative software development

The process of creating software often involves many iterations. In iterative software development, the idea is to implement the basic functionality of a software project and subsequently test to make sure that it works. The next iteration will add more features to the project. Further testing will be done to ensure that the implemented changes work properly; however, retesting the features of the first iteration (all previous iterations, actually) is crucial, as changes made to the application may have inadvertently *broken* things.

Using early prototyping by developing prototypes early in the development process is key in iterative software development. It allows developers to interact frequently with the customer. In fact, iteration is not just performed during coding. It is also performed during analysis, design, prototype generation, and feedback.

For simple programs (where you work alone), iteration typically occurs during design and prototype generation. In the end, the goal is the same: to take small steps towards completion of the project. The end of each step is marked with a period of testing. The goal is to fix errors early. If necessary, earlier design decisions can be revisited.

Generally, robust software requires thoughtful processes to be followed with integrity. The goal is to analyze carefully, specify clearly, design thoroughly, implement and test incrementally, review, revise, and learn. Errors found are treated as successes rather than failures. In fact, there is no such things as error free software!

Errors

Undoubtedly, you will encounter many errors when designing and implementing software projects. Some will be easy to find and fix; others won't. In fact, some will be undetectable (and therefore, never fixed!).

When writing programs, *early* errors are known as **syntax errors**. These are errors that the compiler (or interpreter) spots; as such, they are easy to detect. Fixing them is usually straightforward, because today's compilers are pretty good about locating the source of syntax errors.

Later errors are usually **logic errors**. These are bugs that the compiler cannot spot. In fact, some logic errors are not immediately obvious. Logic errors can be, for example, making a mistake in a calculation (e.g., adding instead of subtracting), making an incorrect assumption about input data, and so on. Logic errors are hard to find because a program still runs even with the logic error. The end result may be incorrect; however, that may not be so obvious (depending on the purpose of the application).

Runtime errors are also later errors that occur during the execution of a program. If your program attempts to open a file that isn't found, for example, it will generate a runtime error. The error doesn't occur if the file is found; therefore, runtime errors may or may not occur (depending on runtime environment).

Most commercial software is loaded with later errors. This begs the question, are errors truly preventable? If not, what can we do about detecting them? Arguably, we need tactics for both preventing and detecting errors. If we operate under the assumption that errors will never be entirely preventable, then we must also consider detecting them. With software engineering techniques, we can lessen the likelihood of errors and improve the chances of error detection.

In software projects, there are several methods of error prevention and detection. **Error reporting** involves providing information about errors that have been detected. But how is this information provided? Is it just printed out to the console? In fact, is there even a human user? Perhaps the application is automated and never executed by a human in front of a computer! A technique, then, may be to send an error message to an object. A better technique may be to return a diagnostic value after completing some action. The preferred technique, however, is to generate exceptions. This will be discussed later in this lesson.

Error handling is the process of dealing with errors programmatically. Should an error occur, the application internally handles it. This requires anticipating failure. A typical point of failure involves obtaining user input. Users vary, and predicting what input they provide is not always possible. Therefore, the sanitation of user input is crucial to preventing many errors. **Sanitizing input** often requires ensuring that the provided input is of the proper type and within the proper range. It usually manifests itself in source code as checking the type and value of arguments passed in to methods.

Exceptions and exception handling

In programming, an **exception** is the interruption of the normal flow of a program. It is something abnormal, yet sometimes expected. Exceptions are a language feature that is predicated on the fact that errors can't be ignored and can often be recovered from.

In Python, exceptions can (and often do) occur. They can be detected and programmatically handled through a **try-except** block. The **try** block identifies statements that, when performed, may cause an exception. The statements in the **except** block are *only* executed if an exception occurs in any of the statements in the try block. It provides code that handles the exception. You may recall this in some of the RPi projects that involved the GPIO pins. The behavior of some of the applications in these projects was to run forever (well, until the user presses Ctrl+C). It is important to ensure that the GPIO pins are reset before terminating a program. Pressing Ctrl+C generates a KeyboardInterrupt exception. The

KeyboardInterrupt exception is defined in Python as some keyboard activity that interrupts the normal flow of the program. Pressing Ctrl+C does exactly this.

Several of the activities that you worked on detected (and handled this exception by cleaning up the GPIO pins). Here is a snippet of code from one such past RPi activity:

```
try:
    while (True):
        # do stuff
except KeyboardInterrupt:
    GPIO.cleanup()
```

The statements in the try block occur forever (i.e., in the `while (True)` loop). Pressing Ctrl+C generates an exception. The except block detects this exception and handles it by cleaning up the GPIO pins. Since the exception is programmatically handled, the interpreter doesn't terminate the program and dump information to the console about the exception. If desired, multiple exceptions can be handled at the same time. In fact, we can create our own exceptions. In Python, these are classes that inherit from (and are subclasses of) the `Exception` class.

Error recovery can be managed by considering a few important things. First, check return values. Implementing a robust return value technique can provide a lot of information about the behavior of a program and aid in detecting and preventing errors. Second, use (and don't ignore) exceptions. Attempt to programmatically handle them. Of course, this means to try to anticipate all possible errors (which is a difficult thing to do). Lastly, include source code to attempt recovery. This may, for example, mean to loop to recover and retry an action that generated an exception. However, an exit strategy may be needed if the error continues to occur. This often happens with input/output (e.g., opening files, accepting user input, etc).

Testing and debugging

A large part of designing software is to ensure that it is free (or nearly free) of errors. This requires testing and debugging. **Testing** searches for the *presence* of errors. It merely indicates that errors exist. Applications are made up of many units. The preferred tactic is to test each unit separately. This is called **unit testing**. Unit testing is performed often during development. The goal is to find errors and fix them early before they become overwhelming.

If ideal coupling and cohesion are maintained, it is easy to identify what a unit should do. This is called its **contract**. Programmers often call this the unit's **interface** (i.e., how it can be interacted with). Most programming languages provide this information via a signature (or header), such as a method's signature. To test units, we often look for potential contract violations. This can be done with positive and negative tests. That is, we test boundaries or boundary conditions (e.g., zero/empty parameters, one, full, etc). We provide values for arguments that aren't necessarily expected to see if we have properly dealt with such an eventuality.

Since testing is often repetitive and time-consuming, it can be automated. In fact, tests are often re-run, over and over again (called **regression testing**) after we have modified a program. Software development professionals often create entire classes whose sole purpose is to test programs. These are sometimes called **test harnesses**.

Through testing, we can detect the presence of errors. Ultimately, however, we must find and fix them. This process is known as **debugging**. Its purpose is the find the *source* of errors. It is important (and

sometimes quite frustrating) to note that errors can occur far (sometimes very, very far) from their source. Debugging helps to lessen the likelihood of errors and to improve the detection of errors.

That being said, it is important to develop good code reading skills and to play “computer.” In fact, there are several techniques that can help with debugging. **Manual walkthroughs** involve getting away from the computer and running through the program by hand (on paper). This is a relatively underused method, but it is useful.

An object's behavior is largely determined by its state. Therefore, incorrect behavior is often the result of incorrect state. Walkthroughs involve tabulating the values of key fields and documenting state changes after method calls. Sometimes, walkthroughs are delivered verbally to other programmers. Sometimes, a method known as **rubber duck debugging** is used. It involves talking to a “bobblehead” or a rubber duck (basically, some sort of inanimate object sitting on one's desk). Verbally explaining source code and the behavior of a program often helps to reveal errors and inconsistencies.

A very popular approach when debugging is to use **print statements** placed in strategic places in the source code. This technique adds output statements throughout the code to show state, position, and sequence. Note that this approach can be overwhelming, as the output can be plentiful. However, the use of one or more debug Boolean variables can help with this. For example:

```
DEBUG = False

...
if (DEBUG):
    print "Something meaningful."
```

Print statements enclosed in an appropriate if block will only be executed if the variable DEBUG is set to true. Sometimes, the need for varying levels of debugging output is needed. In this case, the variable DEBUG can be an integer rather than a Boolean. For example:

```
DEBUG = 3

...
if (DEBUG == 1):
    print "Debug level 1 output"
elif (DEBUG == 2):
    print "Debug level 2 output"
elif (DEBUG == 3):
    print "Debug level 3 output"
```

Most major software projects are designed using IDEs (Integrated Development Environments). You have used IDLE (for Python programs) and the Scratch IDE (for Scratch scripts). Many IDEs integrate a debugger as one part of its functionality (in addition to source code editing, syntax highlighting, compiling or interpreting, etc).

A **debugger** is an interactive debugging tool that helps programmers test and debug their programs. Debuggers are language and environment specific, and they usually work together with the compiler. Debuggers support breakpoints (i.e., setting places in the source code to stop execution). In addition, they provide the ability to step into, through, and over one or more statements. This is useful to see how a single statement (or a group of statements) affects execution, and to identify the source of an error. Lastly, debuggers provide the ability to *watch* and *trace* variables as statements are executed. That is, a

programmer can see the values of variables change as statements are executed. You can see why debuggers are very useful debugging tools.

A few remaining important points, concepts, and ideas

Before leaving this overview of software engineering, there are a few final points to make concerning the development of software projects. Specifically, these refer to generating source code. First, code duplication is an indicator of bad design. Multiple instances of the same snippet of code scattered throughout a program makes maintenance harder and can lead to errors. To remedy this, we can encapsulate the duplicated code in a method that is called when it is needed.

Second, software engineers should practice responsibility driven design. This is the idea that each unit should be responsible for manipulating its own data, and that the unit that owns the data should be responsible for processing it. Intuitively, this leads to loose coupling and localizing change. If and when a change is needed, as few units as possible should be affected.

Third, much of successful software development involves constantly thinking ahead. When designing units, software engineers try to think about likely future changes. Doing so leads to design choices that make it easier to maintain units in the future.

Lastly, refactoring is an integral part of software development and maintaining coupling and cohesion. When units are maintained, new code is often added. This can lead to units becoming longer, responsible for more tasks, and more dependent on other units. Therefore, they may need to be split up into multiple units to maintain ideal coupling and cohesion. This is known as **refactoring**. Of course, units must then be tested (including regression testing).

The game of life

The Game of Life was invented by John Conway in 1970. Generally speaking, it begins with a square 2D matrix of *cells*. Each cell either contains a living organism (or not), forming a population. We sometimes consider the matrix a virtual Petri dish for this reason. The goal is to see how the population evolves over time, according to a few simple rules:

- (1) Any living cell with fewer than two neighbors dies (ostensibly from loneliness);
- (2) Any living cell with more than three neighbors dies (ostensibly from overcrowding);
- (3) Any living cell with exactly two or three neighbors continues living, unchanged, to the next generation; and
- (4) Any dead cell with exactly three neighbors becomes alive (ostensibly from reproduction).

A cell's neighbors are located above, beneath, on either side, and at the diagonals. Therefore, a cell may have up to eight neighbors. The exception, of course, is a cell that is on an edge or in the corner of the matrix. In the table below, the cells highlighted in yellow identify the three neighbors of the cell in the top-left corner of the matrix, the cells highlighted in green identify the five neighbors of the cell in the right-most column, and the cells highlighted in blue identify the eight neighbors of the cell near the bottom-left of the matrix:

*				

Initially, the matrix is randomly populated with living organisms and dead cells. This initial generation is called the **seed**. The rules are then applied for all cells simultaneously. That is, the number of living neighbors for each cell is calculated from the current generation in order to generate the next generation.

Here is an example of the random seed of a 5x5 matrix, forming generation 0:

*	*	*		*
*				
			*	*
	*	*		*
*		*		*

If the rules are applied simultaneously, then the next generation (generation 1) becomes the following:

*	*			
*		*		*
	*	*	*	*
	*	*		*
		*		

As one more example, the next generation (generation 2) becomes the following:

*	*			
*				*
*				*
				*
	*	*	*	

The game continues until either the generation stabilizes and stops changing (becomes a still life), flips between two states called oscillators, or expires into nothing (i.e., all cells are dead). Here's an example of a still life:

	*	*	
	*	*	

No matter how many more generations are produced, the matrix does not change. The following are examples of oscillators (that continuously flip from one to the next, from generation to generation):

		*		
		*		
		*		

	*	*	*	

Coding the game of life

The first issue to tackle is the data structure that will be used to represent the matrix. One obvious choice is to use a list of lists. That is, a list will be used to represent the matrix (or board). Each element of the list will be a list (i.e., a sublist) representing each row of the matrix. Here's an example of the following matrix represented as a Python list:

*	*	*		*
*				
			*	*
	*	*		*
*		*		*

```
[ ['*', '*', '*', '', '*'],
  ['*', '', '', '', ''],
  ['', '', '', '*', '*'],
  ['', '*', '*', '', '*'],
  ['*', '', '*', '', '*'] ]
```

Note that it has been formatted to appear somewhat like the matrix. Python actually represents it on a single line as follows:

```
[ ['*', '*', '*', '', '*'], ['*', '', '', '', ''], ['', '', '', '*', '*'], ['', '*', '*', '', '*'], ['*', '', '*', '', '*']]
```


The first iteration will be to accept the seed from standard input (which can be redirected by the operating system from a file). Additionally, we will just display the board (via a **print** of the list).

The remainder of this lesson has been purposely omitted. The goal is to follow along with your prof as the game of life is implemented in Python using iterative software development and other software engineering principles. Although this is not a particularly difficult or large software project, it is enough to show how software is designed, step-by-step!

The time has come to add another language to our programming “tool belt.” This is a skill that you are going to have to get used to because learning a new programming language is something that a programmer does pretty frequently. Once you have a good understanding of a general purpose programming language (e.g., Python), transitioning to another language is really just a matter of identifying the subtle differences in syntax. It therefore takes a considerably shorter time than learning your first programming language.

Why Java?

Before we answer this question, let's discuss why we need to learn any new programming language in the first place. If Python is as powerful and robust as we've been claiming, why can't we use it for everything? As a programmer, you will spend a lot of your time catering to the needs of a client, and those needs are rarely the same. Sometimes, your tasks will involve maintaining code that was written decades ago in a programming language that is now obsolete. Other times, it will involve working on a small part of a larger system, and the client will require that you work in a language that is consistent with the rest of the system. In these cases, it is important that you not only have some form of experience in multiple languages, but also be able to quickly pick up and use new (to you) programming languages. Additionally, newer languages are always popping up, and you may need to learn them in order to keep relevant in this field.

So back to the original question: Why Java? First, it is considered one of the most widely used programming languages in industry. Second, Java is platform independent. Many programming languages generate compiled executables that are suitable only for the target system on which the code was compiled. This means that the executable code will look different depending on the target system. Java works a bit differently. Java source code is compiled to a platform independent intermediate language known as **bytecodes**. The bytecodes may then be distributed to any target system. To execute the bytecodes, the target system must have a Java virtual machine. One doesn't need access to the original source code to execute a Java program. Instead, only the bytecodes are necessary. Third, Java is secure. In addition to allowing for the distribution of bytecodes without the source code, it protects novice programmers from dealing with many details “under the hood,” and this provides an extra level of security.

How is Java installed?

The instructions for installing Java will vary from system to system as it depends on the operating system (e.g., Windows, Linux, MacOS, etc) and the CPU of the target computer (i.e., 32-bit or 64-bit). There are many online tutorials that show how to install Java on a variety of systems. Google it! Note, however, that you will want the Java Development Kit (JDK). It comes with the tools necessary to compile Java source code to bytecodes. In addition, it comes with the Java Virtual Machine (JVM) that can execute the bytecodes. Do not confuse the JDK with the JRE (Java runtime Environment) that only contains the JVM (i.e., no tools to compile Java source code).

The JDK can be downloaded at: <https://java.com/en/download/>.

A first Java program

When you were introduced to Python, the concept of an IDE (Integrated Development Environment) was also covered. An IDE is a form of editor that allows you to write programs in one or more

programming languages. It normally has features such as syntax highlighting, syntax suggestions, and allows you to compile and/or execute the programs that you have written easily. For Python, the default IDE on the RPi is IDLE. Unfortunately, IDLE only works with Python.

There are IDEs that are more general purpose than IDLE, in that they support more than one programming language. Examples include Eclipse, NetBeans, and Code::Blocks. The process of installing and setting up these IDEs is beyond the scope of this lesson. If you want to work on large projects in Java and other general purpose programming languages, you will want to research how to install a robust IDE on your system. As an FYI, Eclipse is perhaps the most widely used IDE for Java. For the purpose of this lesson, it is significantly easier to use a general purpose text editor, and compile Java source code and execute the resulting Java bytecodes at the command line (i.e., in the terminal or console).

One of the main features of Java is that it is fully object-oriented. This means that everything that you do within Java has to be defined in a class. It embodies the principle that everything is an object! Even a simple hello world program must be in a class. Unlike Python, where indentation is used to identify blocks of statements, Java used curly braces. The left (or open) curly brace is typically placed after the header of a block and shows where the block begins, and the right (or close) curly brace shows where the block ends. Here's an example:

```
class HelloWorld
{
    ...
}
```

Note that there is still indentation within the block. Although this is not required, it is considered good programming practice to do so. In fact, not doing so is indicative of a worthless programmer. Well, maybe that's too harsh. Maybe not...

You'll notice the keyword `class`, which signifies that the word coming after it is the name of a class. This is similar to Python. The name of the class can be any identifier. The only restrictions on identifiers is that they have to be made up of alphanumeric characters and underscores, and cannot begin with a digit. It is Java convention to initially capitalize class names; however, that's not enforced. Since it is convention, however, it does make your code easier to read by professional programmers.

Unlike Python, where statements don't necessarily have to be encapsulated in methods, Java requires everything to be in at least one method. Method signatures have the following form:

```
modifier return_type method_name(parameter_list)
{
    method body
}
```

The **modifier** is a word that determines the accessibility (or usability) of a method. The most common modifiers are:

- `public`: indicates that the method can be accessed from anywhere; and
- `private`: indicates that the method can only be accessed within its defining class.

The **return_type** is a key word that informs the compiler of the kind of result that the function returns to whichever statement called the function in the first place. In Python, there is no such thing as formally

defined data types. A Python variable can take on any type, and a Python function can return any type (even multiple types). In Java, variables must be declared along with their type. These are valid throughout the duration of the program and cannot be changed! Functions may return a value, and if so, the return type must be specified. If the function does not return anything, then the keyword `void` is used. Common return types are `int`, `long`, `float`, `double`, `String`, and `char`. We will cover some of these later in this lesson. The return type can also be a class. That is, the value returned can be an instance of a class!

The **method_name** can be any valid Java identifier. It is convention that it begin with a lower case character, and that successive words be initially capitalized (e.g., `printGrades`). This way, it is easy to differentiate between a class name and a function name.

The **parameter_list** is a comma separated list of variables and their types that are passed in as inputs to the function. If a function does not require any parameters, then the parentheses are left empty. Below are some examples of method signatures:

- **public** `int` `getSum(int a, int b)`
- **public** `String` `printBoard()`
- **private** `double` `getAverage(double a, double b, double c)`
- **private** `void` `doSomethingCool(int a, double c, String b, char d)`

Since a program's source code in Java must be specified within at least one function, how does the compiler know which function is to be executed first? There is a special function with a special name that is always executed first in a Java program. That function has a specific signature:

```
public static void main (String[] args)
```

Note that the method is `public`, and can therefore be accessed outside of the class. It does not return anything due to the keyword `void`. Its name is `main`, and it takes an array of `Strings` called `args`. The square brackets after the `String` type indicates that it is an array of `Strings`. Arrays in Java are similar to lists in Python. The `static` keyword enforces the idea that, even if multiple objects were created from the class in which the `main` function is located, there will only be one copy of `main` that is the same for all the objects. That is, the `main` function is shared among all instances of the class that contains it. The `main` function is considered the entry point of a program.

Let's put all of this information together to create our first “hello world” Java program:

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Programming rules, man!");
    }
}
```

Note the semicolon at the end of the statement that displays “Programming rules, man!” In Java (and, in fact, many other programming languages), the end of a statement is marked by a semicolon.

Did you know?

A missing semicolon at the end of a Java statement is the most common syntax error. It is practically guaranteed that you will forget to put a semicolon at the end of a Java statement at some point.

Let's discuss the content of the source code to the simple hello world program. There is really only a single statement in the program: the one that displays the string. It is enclosed in the main function, which itself is enclosed in the HelloWorld class. Let's formally discuss the single statement:

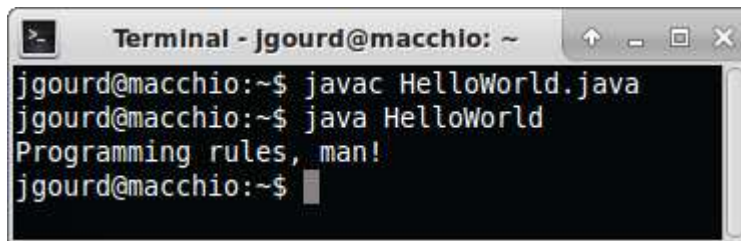
```
System.out.println("Programming rules, man!");
```

In order to access the members of classes, we use the dot notation. Recall that we used the dot notation to access members within classes in Python as well. In Java, the `println` function displays the parameter specified to it to standard output (i.e., to the console). To access it, we fully qualify it with `System.out.println`. Technically, `System` is a class, inside of which is an object called `out`. The defining class of the `out` object provides the `println` function.

Note that the `println` function automatically inserts a newline after displaying its parameter to the console. This is similar to the behavior of the `print` statement in Python. To suppress the newline, use the function `print` instead of `println`. For example:

```
System.out.print("Programming rules, man!");
```

So how do we compile and execute our first Java program? Here is this process from the command line:

A screenshot of a terminal window titled "Terminal - jgourd@macchio: ~". The terminal shows the following commands and output:

```
jgourd@macchio:~$ javac HelloWorld.java
jgourd@macchio:~$ java HelloWorld
Programming rules, man!
jgourd@macchio:~$
```

On some operating systems (e.g., Windows) it may be necessary to include the location of the Java compiler and other executables in the *path* environment variable in order to allow compilation and execution from any folder. To compile a Java program from the command line, we use the following syntax:

```
javac myfile.java
```

Note that it is convention to name our Java source files with a `.java` extension. The compilation process generates a new file of the same name; however, it has a `.class` extension. To execute the compiled Java bytecodes, we use the following syntax:

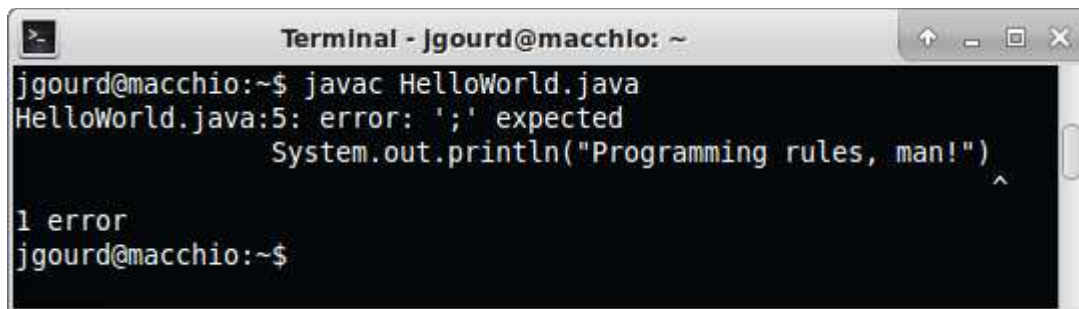
```
java myfile
```

Note that we do not specify the `.class` extension when executing the bytecodes!

Here's an example of what happens when you try to compile a Java program with a syntax error. First, here's a modified version of the hello world program:

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Programming rules, man!")
    }
}
```

Did you notice the error? Here's the output generated when compiling this modified program:

A screenshot of a terminal window titled "Terminal - jgourd@macchio: ~". The terminal shows the command "javac HelloWorld.java" being executed. The output is "HelloWorld.java:5: error: ';' expected" followed by "System.out.println(\"Programming rules, man!\")" with a caret pointing to the end of the line. Below this, it says "1 error" and "jgourd@macchio:~\$".

```
Terminal - jgourd@macchio: ~
jgourd@macchio:~$ javac HelloWorld.java
HelloWorld.java:5: error: ';' expected
    System.out.println("Programming rules, man!")
                                ^
1 error
jgourd@macchio:~$
```

The error is actually quite easy to see when the program is compiled. The Java compiler does a pretty good job of informing us what and where the error is: a missing semicolon at the end of the statement that displays output to the console.

Data types, constants, and variables

Although Python does not support the formal declaration of variables with data types, the data types in Java are actually very similar to those in Python. Java supports primitive types such as integers (`int`), long integers (`long`), single precision floating point numbers (`float`), double precision floating point numbers (`double`), characters (`char`), Booleans (`boolean`), and more. In addition it supports more complex types such as a `String` (which, as in Python, is a *string* of characters) and user-defined types defined in classes that can be instantiated as objects.

When it comes to the variables and the way that they are used, there is one difference with Java. In fact, this difference was something that we dealt with in Scratch. Before a variable can be used, it has to be declared. This means that the compiler needs to know what data type and name will be associated with a variable. Once it has been declared as one type, it can not be reused to store a different data type.

Declaring a variable can be done with a statement in the following format:

```
data_type variable_name;
```

The statements below are examples of valid variable declarations:

- `int x;`
- `float num1, num2;`
- `double c;`
- `String str;`
- `char c;`

- `boolean b;`

Note the second example. Here, two variables are actually declared, `num1` and `num2`, both of the data type `float`. Once declared, the variables can be assigned and used as they would be in Python (i.e., they do not have to be redeclared every time we wish to use one). In fact, we can assign new values to previously declared (and even previously initialized) variables at any point in a program.

It is also possible to declare and initialize a variable with a value in a single Java statement, as shown in the examples below:

- `int x = 3;`
- `float num1 = 3.14, num2 = 1.414;`
- `double c = 2.78;`
- `String str = "hard drive";`
- `char c = 'p';`
- `boolean b = false;`

It is also possible to assign the result of an expression to a variable, even as you declare it:

- `double d = 1.3 + 2.5;`
- `char grade = calcGrade();`

Note that the function `calcGrade` in the second example must return a value of type `char`.

Did you know?

While the character `#` is used to mark single-line comments in Python, Java uses `//`. For example:

```
// this is a comment
```

Java also uses `/*` to begin a multi-line comment and `*/` to end a multi-line comment. For example:

```
/* this is
   a multi-line
   comment */
```

Of course it is possible to use `//` for a comment spanning multiple lines so long as each line begins with `//`.

Now that we have a general understanding of primitive data types and how to use them, let's write a slightly more complicated program that displays the sum of two integers to the terminal.

```
public class Sum
{
    public static void main(String[] args)
    {
        int a = 5;
        int b = 7;

        System.out.println(a + " + " + b + " = " + (a + b));
    }
}
```

Here is the output of this program's compilation and execution:

A screenshot of a terminal window titled "Terminal - jgourd@macchio: ~". The terminal shows the following commands and output:

```
jgourd@macchio:~$ javac Sum.java
jgourd@macchio:~$ java Sum
5 + 7 = 12
jgourd@macchio:~$
```

Let's look more closely at the following statement:

```
System.out.println(a + " + " + b + " = " + (a + b));
```

The variable *a* is an integer. It appears to be added to the string "+ ". However, Java cannot add an integer to a string; therefore, it converts the integer to a string first and then concatenates the two strings. Since the value of *a* is 5, then *a* + "+ " results in "5" + "+ " which yields the string "5 + ". In this statement, the integers are all converted to strings first before concatenating them together.

Constants in Java are declared and initialized as follows:

```
public static final int NUM_RECORDS = 100;
```

Constants in Java are public (i.e., accessible everywhere), static (i.e., one value exists for all instances of the class), and final (i.e., their value will not change). They must also have a type and a name. It is convention to capitalize the entire constant name and use an underscore to separate words. Of course, the constant must be assigned a value.

Activity 1: What's my grade?

Let's write a simple program that calculates the letter grade corresponding to a student's score on a test. The program will contain two functions. The bulk of the code will be written in the main function; however, it will take advantage of a `calcGrade` function that takes the score (a double) as its parameter and returns the matching letter grade. Here is the source code:

```
public class LetterGrade
{
    public static void main(String[] args)
    {
        double score = 87.5;
        char letter_grade;

        letter_grade = calcGrade(score);

        System.out.println("Your grade is " + letter_grade);
    }

    public static char calcGrade(double d)
    {
        if (d >= 90.0)
```

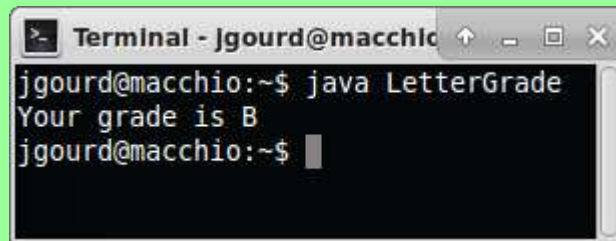


```

        return 'A';
    else if (d >= 80.0)
        return 'B';
    else if (d >= 70.0)
        return 'C';
    else if (d >= 60.0)
        return 'D';
    else
        return 'F';
    }
}

```

And here is the output:



```

Terminal - jgourd@macchio
jgourd@macchio:~$ java LetterGrade
Your grade is B
jgourd@macchio:~$

```

Although the syntax is different, you should be able to use your existing knowledge of Python in order to understand what is going on. In Java, we use `else if` instead of `elif` to add a condition to the `else` clause of an `if` statement. In fact, we can chain them (just as we do with `elif` in Python). Note that Java does not require a colon at the end of an `if` statement.

If the true or false (else) part of an `if` statement only includes a single statement, then curly braces can be omitted. By default, a block contains a single statement. If, however, multiple statements were required, curly braces would be needed. Here's an example of a multi-line `if else` statement:

```

if (d >= 90)
{
    System.out.println("Wow, good job!");
    return 'A';
}

```

Compare this with a similar block of statements in Python:

```

if (d >= 90):
    print "Wow, good job!"
    return 'A'

```

Input and output

You have already seen how output can be done in Java using `System.out.print` and `System.out.println`. This is the standard way to direct output to standard output. To obtain input from standard input, we can use `System.in` (the inverse of `System.out`).

Although it is not as straightforward, obtaining user input can be accomplished as follows:

```
import java.util.Scanner;

public class Input
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);

        while (s.hasNextLine())
        {
            String line = s.nextLine();

            System.out.println(line);
        }
    }
}
```

First, note that the `import` keyword is also used in Java to bring in external libraries. In fact, it's not much different than in Python! The statement `import java.util.Scanner` imports the `Scanner` class from the `util` library in Java. To setup user input, we instantiate a new `Scanner`, using `System.in` as the parameter (i.e., we can now scan for user input from standard input). Note the use of the keyword `new`. In Java, the keyword `new` is used to create a new instance of a class. The `Scanner` class has several useful methods. The first that we will make use of is `hasNextLine`, which returns true if the scanner detects that another line of input exists. The second is `nextLine`, which returns the next line of input. This simple program obtains user input and echoes it back to the console.

Operators

For the most part, the operators that you learned in Python are the same (or very similar) in Java as well. This means that it should be relatively straightforward to convert code that you've written in Python to Java. There are, however, a few exceptions that we will now discuss.

The typical **arithmetic operators** for addition (+), subtraction (−), multiplication (*), division (/), and modulus (%) are exactly the same in Java. Java, however, does not use `**` as the exponentiation operator. Instead, it makes use of a math library. Specifically, a function called `pow` can be called as follows to replicate the expression `2 ** 3` in Python:

```
Math.pow(2, 3);
```

Another difference in the arithmetic operators is the floor division. Python has a floor division operator (`//`). As you have seen, this is used in Java to denote a single line comment. To obtain floor division in Java, we need to make use of the math library again, as follows:

```
Math.floor(5.2 / 8.11);
```

Note that, if the values that you are dividing are both integers, (i.e., whole numbers), then the result of their division will be an integer! This is the same result as a floor division. In fact, this is something that you should be very careful about in Java, because it is often the source of logical errors that are very difficult to trace. Java's division operator will give different results for integers than it would for floating point numbers. Take a look at the examples below:

- `1 / 2 → 0`
- `1.0 / 2 → 0.5`
- `1 / 2.0 → 0.5`
- `1.0 / 2.0 → 0.5`
- `(float)1 / 2 → 0.5`
- `1 / (float)2 → 0.5`

The first example is a division of integers which will always return an integer result that is equal to the quotient of the division. If the remainder is needed, we can use the modulus operator. The second, third, and fourth examples are divisions which involve a floating point value as one (or both) of the operands. In this case, the division operator returns the expected floating point result. The fifth and sixth examples result in floating point division because one of the operators is converted or **typecast** to a floating point value. Typecasting is used to inform the compiler that the variable that it applies to is to be considered of the type specified in the parentheses. Depending on the direction of the typecast, typecasting can result in a loss or increase in accuracy of the operation.

The following table enumerates all of Java's arithmetic operators (assume that `a = 23`, `b = 17`, `c = 4.0`, and `d = 8.0`):

Java Arithmetic Operators and Examples			
+	addition	<code>a + b = 40</code>	<code>c + d = 12.0</code>
-	subtraction	<code>a - b = 6</code>	<code>c - d = -4.0</code>
*	multiplication	<code>a * b = 391</code>	<code>c * d = 32.0</code>
/	division	<code>a / b = 1</code>	<code>c / d = 0.5</code>
%	modulus	<code>a % b = 6</code>	<code>c % d = 4.0</code>

All but one of the Python **relational operators** apply in Java. Recall that Python provides two relational operators to check for inequality: `!=` and `<>`. Java does not support the latter (i.e., it only supports `!=`). The following table enumerates all of Java's relational operators (assume that `a = 23` and `b = 17`):

Java Relational Operators and Examples		
<code>==</code>	equality	<code>a == b</code> is false
<code>!=</code>	inequality	<code>a != b</code> is true
<code>></code>	greater than	<code>a > b</code> is true
<code><</code>	less than	<code>a < b</code> is false
<code>>=</code>	greater than or equal to	<code>a >= b</code> is true
<code><=</code>	less than or equal to	<code>a <= b</code> is false

Java has special **prefix and postfix operators** that you most likely have not seen before (certainly not in Python, anyways). These operators are often referred to as syntactic sugar, in that they provide a shorthand for incrementing or decrementing a variable. Here is a snippet of code as an example:

```
int p = 3;
p++;    // p is now 4
```

```

++p;    // p is now 5
int q = 5;
q--;    // q is now 4
--q;    // q is now 3

```

The ++ and -- operators are used to increment and decrement a variable respectively. They only increment or decrement by 1 in each case. The location of the ++ or -- operator relative to the variable determines the order in which this increment or decrement is done and is dependent on the context in which it is used. If the operator is placed before the variable (i.e., as a prefix), then it means that the increment or decrement is carried out before the variable is used. Conversely, if the operator is placed after the variable (i.e., postfix), then it means that the increment or decrement is carried out after the variable is used. Here's an example of the nuances between prefix and postfix:

```

int p = 3;
System.out.println(p++);    // p is displayed and then
                             // incremented to 4

System.out.println(p);
System.out.println(++p);    // p is incremented to 5 and
                             // then displayed

```

And here's the output:



```

Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java Arithmetic
3
4
5
jgourd@jgourd-latech:~$

```

Note that the following statements all accomplish the same thing: increment the value of the variable *p* by 1:

- `p = p + 1;`
- `p += 1;`
- `p++;`
- `++p;`

The first two are also valid in Python (and do the same thing), although no semicolons are required.

Java **assignment operators** are similar to those in Python, except that there is no assignment operator for exponentiation or floor division. The following table enumerates all of Java's assignment operators (assume that $a = 23.0$ and $b = 17$):

Java Assignment Operators and Examples		
=	$a = b$ assigns b to a	$a = 17.0$
+=	$a += b$ increments a by b (same as $a = a + b$)	$a = 40.0$
-=	$a -= b$ decrements a by b (same as $a = a - b$)	$a = 6.0$
*=	$a *= b$ multiplies a by b and stores the result in a (same as $a = a * b$)	$a = 391.0$
/=	$a /= b$ divides a by b and stores the result in a (same as $a = a / b$)	$a = 1.3529411764705883$
%=	$a \% = b$ divides a by b and stores the remainder in a (same as $a = a \% b$)	$a = 6.0$

In Java, **bitwise operators** work exactly as they do in Python, on bits and perform bit-by-bit operations. The following table enumerates all of Java's assignment operators (assume that $a = 60$, or 00111100 in binary, and $b = 13$, or 00001101 in binary):

Java Bitwise Operators and Examples		
&	bitwise <i>and</i>	$a \& b = 00001100$ (or 12 in decimal)
	bitwise <i>or</i>	$a b = 00111101$ (or 61 in decimal)
^	bitwise <i>xor</i>	$a \wedge b = 00110001$ (or 49 in decimal)
~	bitwise <i>not</i>	$\sim a = 11000011$ (or -61 in decimal; we will explain this one later)
<<	left shift	$a \ll 2 = 11110000$ (or 240 in decimal)
>>	right shift	$a \gg 2 = 1111$ (or 15 in decimal)

In Python, we had words to represent the **logical operators** *and*, *or*, and *not*. In Java, we have symbols to represent these same operators. Logical operators operate on conditions and provide the overall logical result. The following table enumerates all of Java's logical operators (assume that a is true and b is false):

Java Logical Operators and Examples		
&&	logical <i>and</i>	$a \&\& b$ is false
	logical <i>or</i>	$a b$ is true
!	logical <i>not</i>	$!a$ is false; $!b$ is true

The logical operators are most often used with expressions in the conditions of if statements and loop constructs.

Primary control constructs

Similar to Python, Java uses **sequence** as a primary control construct. Statements are executed, one after another. Control flow is as expected. It can, however, be changed when, for example, functions are called.

Selection in Java is very similar to Python and is implemented as an if statement. As shown earlier, if statements are nearly the same; however, no colon is used at the end of the if statement, braces are used to denote blocks, and `else if` is used instead of `elif`. Note that the conditions of if statements must be enclosed in parentheses:

```
if (grade > 89.5)
    letter_grade = 'A'
else if (grade > 79.5)
    letter_grade = 'B'
else if (grade > 69.5)
    letter_grade = 'C'
else if (grade > 59.5)
    letter_grade = 'D'
else
    letter_grade = 'F'
```

In Python, strings can be enclosed in single or double quotes. In Java, however, they must always be enclosed in double quotes; for example:

```
"This is a string."
```

In Java, characters are always enclosed in single quotes; for example:

```
'^'
```

There is another selection construct in Java called a switch-case. Let's explain it by starting with an example:

```
char grade = 'B';

switch (grade)
{
    case 'A':
        System.out.println("Excellent job!");
        break;
    case 'B':
        System.out.println("Great job!");
        break;
    case 'C':
        System.out.println("Good job!");
        break;
    case 'D':
        System.out.println("Fair job!");
        break;
    case 'F':
        System.out.println("Poor job!");
        break;
    default:
        System.out.println("Invalid grade!");
        break;
}
```

A switch statement is very much like an if-else with many possible else-if clauses. The condition is restricted to variables of the type `int`, `char`, and `String` (among a few others that are not used often). In the example above, the variable that serves as the condition value is *grade*. If the grade is A, then the first case is *triggered* (and the string “Excellent job!” is echoed to the console). The `break` statement then exits the switch construct so that none of the other cases are executed. In a switch statement, cases can *fall through* each other. Here's an example, where earning an A or B yields the string “Great job!”, a C yields the string “Good job!”, and a D or F yields the string “Poor job!”:

```
char grade = 'B';

switch (grade)
{
    case 'A':
    case 'B':
        System.out.println("Great job!");
        break;
    case 'C':
        System.out.println("Good job!");
        break;
    case 'D':
    case 'F':
        System.out.println("Poor job!");
        break;
    default:
        System.out.println("Invalid grade!");
        break;
}
```

The default case is a *catch-all* that executes if none of the other cases are evaluated to be true. If, for example, the grade provided was mistakenly entered as 'E', then the output would be “Invalid grade!”

Repetition in Java is somewhat similar to Python. There are three repetition constructs. The while-do loop is similar to Python. It is formatted as follows:

```
while (condition)
{
    loop_body
}
```

Note that the braces are only required if there is more than one statement in the loop body. Here's an example of a single statement in the loop body that does not require braces:

```
int sum = 0;
while (sum > 0)
    sum--;
```

In the example above, the loop body is never executed because the variable *sum* is initialized to 0. There is a variation of the while-do loop called the do-while loop that is formatted as follows:

```
do
{
    loop_body
} while (condition);
```

The main differences of this form of the while loop are that (1) the loop body always executes at least one time; and (2) there is a semicolon after the condition at the bottom of the loop. Here's an example:

```
int sum = 0;
do
{
    sum--;
} while (sum > 0);
```

In this example, the loop body executes once. When it does, the value of the variable *sum* is decremented by one (to -1), and the condition at the bottom of the loop becomes false (which breaks out of the while loop).

In Java, the for loop is drastically different than it is in Python. It is formatted as follows:

```
for (initialization; condition; change)
{
    loop_body
}
```

The initialization portion is used to initialize one or more variables (usually used as a counter). The condition portion is evaluated and decides if the loop body is to be executed (if the condition evaluates to true). The change portion changes the value of one or more of the variables in the initialization portion (e.g., increment or decrement). This occurs after each execution of the loop body. Here's an example:

```
int i;

for (i=0; i<10; i++)
{
    System.out.print((i * i) + " ");
}
```

This example changes the value of the variable *i* from 0 to 10. The loop body is executed when *i* is between 0 and 9 inclusive. When its value is 10, the loop body is not executed. Initially, the value of *i* is initialized to 0. Since 0 is less than 10, the loop body is executed. After this, *i* is incremented (to 1). Since 1 is less than 10, the loop body is executed again. This continues. At one point, *i* has the value 9. Since 9 is less than 10, the loop body is executed. After this, *i* is incremented (to 10). Since 10 is not less than 10, the for loop is then terminated, and control continues to the statement that follows it.

Formally, the initialization is done before the first iteration. The condition is evaluated before each iteration and determines whether the loop body is executed. The change is carried out at the end of each iteration. While the majority of your for loops will have initialization, condition, and change dealing with the same variable, this is not a rule. The change portion can vary greatly. Sometimes, it represents

an increment (as in the example above). At other times, it could be a decrement or even an entirely different arithmetic expression. The condition portion, however, should always evaluate either to true or false. The for loop is exited when the condition evaluates to false.

Additionally, it is possible to leave some or all of the parts of the for loop (i.e., initialization, condition, and change) empty. There must always be semicolons that delineate the three portions of the for loop, however, even if some are empty. If one selects to have one or more of the portions empty, a way to cater for the missing portion(s) must be specified. Here's an example:

```
int i = 10;

for (;;)
{
    i--;
    if (i == 0)
        break;
}
```

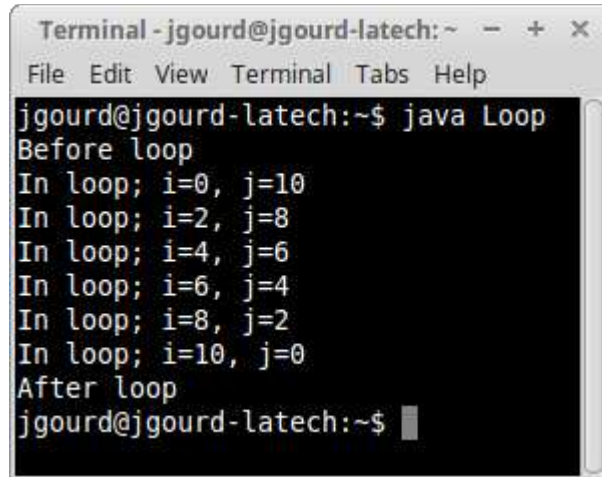
The for loop header, `for (;;)`, is technically an infinite loop. Therefore, there must be some way to exit the loop in the loop body. This is accomplished as the variable *i* is decremented at each iteration of the loop. It will eventually reach the value 0; and when that occurs, the condition of the if statement will evaluate to true, and the `break` keyword will terminate the loop.

Lastly, it is also possible to place more than one initialization in the initialization portion, more than one condition in the condition portion, and more than one change in the change portion. The only requirement is that the different components are separated by a comma. Also, variables can actually be declared in the initialization portion, eliminating the need to do so beforehand. Here is an example of both of these:

```
System.out.println("Before loop");
for (int i=0, j=10; i<=100 && j >= 0; i+=2, j-=2)
{
    System.out.println("In loop; i=" + i + ", j=" + j);
}
System.out.println("After loop");
```

Note that, if variables are declared in the initialization portion of the for loop, then they become local to the loop and are unavailable after the loop. Let's explain the snippet of code above. First, the variables *i* and *j* are declared and initialized to 0 and 10 respectively in the initialization portion of the for loop. Second, if *i* is less than or equal to 100 and *j* is greater than or equal to 0, then the loop body is executed. Third, after the loop body is executed, *i* is incremented by 2 and *j* is decremented by 2.

Here is the output:



```
Terminal - jgourd@jgourd-latech:~ - + X
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java Loop
Before loop
In loop; i=0, j=10
In loop; i=2, j=8
In loop; i=4, j=6
In loop; i=6, j=4
In loop; i=8, j=2
In loop; i=10, j=0
After loop
jgourd@jgourd-latech:~$
```

There is another variation of the for loop, called the foreach loop, that will be discussed later in this lesson.

Variable scope

In Java, scoping of variables is similar to Python. That is, global variables can be declared. They must be declared in the class, but outside of any function in the class. This makes them class variables (which you already know about). Technically, they are global to the class. Variables declared in functions are local to those functions.

Program flow

Let's use a previous example to illustrate program flow in Java (which, by the way, is similar to Python):

```
public class LetterGrade
{
    public static void main(String[] args)
    {
1:         double score = 87.5;
2:         char letter_grade;

3:         letter_grade = calcGrade(score);

4:         System.out.println("Your grade is " + letter_grade);
    }

    public static char calcGrade(double d)
    {
5:         if (d >= 90.0)
6:             return 'A';
7:         else if (d >= 80.0)
8:             return 'B';
9:         else if (d >= 70.0)
10:            return 'C';
11:        else if (d >= 60.0)
```

```

12:         return 'D';
13:     else
14:         return 'F';
    }
}

```

Using the source code above, in the space below try to number the statements in the order that they would be executed:

Note that the main function doesn't necessarily have to be placed as the first function in a class. Functions can be placed in any order! Here's a slightly modified version of the program above:

```

public class LetterGrade
{
    public static char calcGrade(double d)
    {
1:         if (d >= 90.0)
2:             return 'A';
3:         else if (d >= 80.0)
4:             return 'B';
5:         else if (d >= 70.0)
6:             return 'C';
7:         else if (d >= 60.0)
8:             return 'D';
9:         else
10:            return 'F';
    }

    public static void main(String[] args)
    {
11:        double score = 66.0;
12:        char letter_grade;

13:        letter_grade = calcGrade(score);

14:        System.out.println("Your grade is " + letter_grade);
    }
}

```

Using the modified source code above, in the space below try to number the statements in the order that they would be executed:

Arrays

By now, you are familiar with Python lists. In previous lessons, we mentioned that a list in Python is similar to an array. The main differences are that (1) a Python list can store heterogeneous pieces of data, while arrays are designed to store homogeneous pieces of data (i.e., values of the same data type); and (2) arrays must be declared with a fixed capacity (i.e., they cannot grow as needed).

Java does not support lists in the same way that Python does. Instead, it supports arrays as described above. In the end, they are almost the same. Declaring an array in Java is very similar to declaring any other data type. The difference is that square brackets are placed between the data type and the identifier (i.e., the variable name). Here is an example that declares two variables: (1) an integer, `x`; and (2) an array, `y`:

```
int x;  
int[] y;
```

Just as with simple variables, arrays can be declared and initialized in the same statement:

```
int x = 10;  
int[] y = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

The second statement in the example above declares an array of integers and initializes it with the integers 1 through 10 at indexes 0 through 9 respectively. This sets the array's capacity at 10 values. The curly braces allow for the initialization of arrays upon declaration.

Once arrays have been declared, its individual elements can be accessed in a manner similar to accessing the elements of a list in Python (i.e., using the index in square brackets following the name of the array).

Here's an example:

```
System.out.println(y[3]);    // outputs 4  
y[6] = -1;    // array is now { 1, 2, 3, 4, 5, 6, -1, 8, 9, 10 }
```

Note that array slices are not supported in Java. This is rather unfortunate! It is also possible to just declare an array and subsequently initialize it as follows:

```
int[] y = new int[10];  
  
for (int i=0; i<10; i++)  
    y[i] = i * i;
```

This snippet of code declares an array of 10 integers. It then iteratively populates the array with the first ten squares (i.e., $0 * 0$, $1 * 1$, $2 * 2$, etc).

Here are a few other differences between Python lists and Java arrays:

- There is no way to actually delete an element from an array (instead, we can zero out an element or shift elements over to effectively delete an element); and
- Similarly, there is no way to insert an element in an array (once the capacity is set, it cannot be changed).

Can you understand what the source code below produces and how it works?

```
public class Arrays
{
    public static void main(String[] args)
    {
        int[] test1 = { 67, 23, 98, 45, 99, 45 };
        int[] test2 = { 88, 44, 94, 78, 64, 44 };
        int i = 0;

        while (i < 6)
        {
            if (test1[i] > 80 && test2[i] > 80)
            {
                System.out.println("Student " + i +
                    " doesn't need to take the final");
            }
            else if (test1[i] > 80 || test2[i] > 80)
            {
                System.out.println("Student " + i +
                    " can decide to take the final");
            }
            else if (!(test1[i] > 60) && !(test2[i] > 60))
            {
                System.out.println("Student " + i +
                    " definitely needs to take the final");
            }
            else
            {
                System.out.println("Student " + i +
                    " -- no decision");
            }

            i++;
        }
    }
}
```

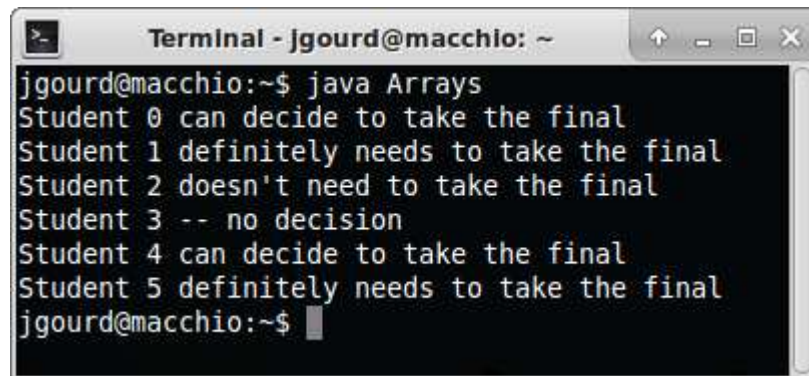
You will notice that statements in Java can be split among multiple lines without needing any special formatting or continuation character. This is evident in the statements outputting strings to the console via `System.out.println`. In Python, a backslash is required to inform the interpreter that a statement is continued on the next line.

The program first declares and initializes two arrays of six integers. The first, `test1`, represents six students' grades for the first test. The second, `test2`, represents the same six students' grades for the second test. The while loop then iterates over all of the students' grades (there are two of them for all six students). Next, an informative message is displayed depending on the test grades of each student. If a student earned more than an 80 on both tests, then the student doesn't need to take the final. If a student earned more than an 80 on one of the tests but not the other, then the student can decide to take the final.

If a student did not earn more than a 60 on both tests (i.e., earned 60 or less on both tests), then the student must take the final. If none of these cases is true, then there is no decision for the student.

The first student, represented by the values in the arrays at the first index (i.e., index 0), earned a 67 on the first test and an 88 on the second test. Following the logic, this student “can decide to take the final.” The second student, represented by the values in the arrays at the second index, earned a 23 on the first test and a 44 on the second test. Following the logic, this student “definitely needs to take the final.” The third student earned a 98 and a 94. This student “doesn't need to take the final.” The fourth student earned a 45 and a 78, which results in “no decision.” The fifth student earned a 99 and a 64, and “can decide to take the final.” The sixth student earned a 45 and a 44, and “definitely needs to take the final.”

To confirm this logic, here is the output:

A screenshot of a terminal window titled "Terminal - jgourd@macchio: ~". The window shows the execution of a Java program named "Arrays". The output displays the decision for six students based on their test scores. The text in the terminal is as follows:

```
jgourd@macchio:~$ java Arrays
Student 0 can decide to take the final
Student 1 definitely needs to take the final
Student 2 doesn't need to take the final
Student 3 -- no decision
Student 4 can decide to take the final
Student 5 definitely needs to take the final
jgourd@macchio:~$
```

The foreach loop

Earlier, the foreach loop was briefly mentioned as another variation of the for loop. Formally, it allows iteration over the items in any type of collection (e.g., an array). Java has many collections, and discussing them is beyond the scope of this lesson. However, let's take a look at how the foreach loop can be used to iterate over the items in an array. Here's an example:

```
double[] cool_constants = new double[5];

cool_constants[0] = 3.14159; // pi
cool_constants[1] = 2.71828; // e
cool_constants[2] = 1.414;   // sqrt(2)
cool_constants[3] = 1.73;    // sqrt(3)
cool_constants[4] = 1.618;   // phi

for (double d : cool_constants)
    System.out.println(d);
```

The foreach loop actually uses the keyword `for` (and not literally `foreach`). The portion inside the parentheses contains two parts, separated by a colon. The first is a data type and variable that will represent each of the values in the array. The second is the array name. The colon can be read as “in.” In fact, the entire foreach loop above can be read as, “For each double value temporarily called *d* in the array `cool_constants`.” The body of the foreach loop just displays the value temporarily stored in the variable *d*.

The foreach loop is a neat way to iterate over the items in an array, where each item is temporarily stored in a variable. Note that any index (i.e., location) information is not available in a foreach loop. That is, the location of the values in the array is not provided. If the index was required, then a for loop would be needed. For example:

```
for (int i=0; i<cool_constants.length; i++)  
    System.out.println(cool_constants[i]);
```

You may have noticed the condition in the for loop, `i<cool_constants.length`. This brings up the useful fact that the length of an array can be obtained by accessing its `length` member. This should clue you in that arrays must therefore be objects and have a defining class! The `length` member turns out to be an instance variable defined in the `Array` class.

Activity 2: Selection sort followed by a binary search

In this activity, we will implement the selection sort on an array, and subsequently search the sorted array for a specified value using the binary search.

Since you should be quite familiar with both the selection sort and the binary search, we will skip their mechanics and just show their algorithms. First, recall the selection sort:

```
n ← length of the list  
for i ← 0..n-2  
    minPosition ← i  
    for j ← i+1..n-1  
        if item at j < item at minPosition  
        then  
            minPosition ← j  
        end  
    next  
    swap items at i and minPosition  
next
```

The selection sort maintains sorted and unsorted portions of the list. It repeatedly places the smallest value in the unsorted portion of the list at the end of the sorted portion (or the beginning of the unsorted portion) of the list. The algorithm stops when the unsorted portion of the list is empty.

Second, recall the binary search:

```
num ← number to search for  
first ← 0  
last ← number of items in the list - 1  
while first ≤ last  
    mid ← floor((first + last) / 2)  
    if num = item at mid of the list  
    then  
        return mid  
    else if num > item at mid of the list  
    then  
        first ← mid + 1  
    else
```

```

        last ← mid - 1
    end
    return -1

```

The binary search selects the middle value in the current portion of the list (initially, the entire list) and compares it to the specified value. If the values match, then the (middle) index is returned. If not, and the specified value is greater than the value in the middle, then the beginning of the list is shifted to the element to the right of the middle of the list. The rest of the list is effectively discarded. Otherwise, the specified value is less than the value in the middle, and the end of the list is shifted to the element to the left of the middle of the list. Note that the algorithm has been slightly modified to return the index of the specified value (if it is found in the list) or -1 (if it isn't found in the list).

Let's write a program that generates an array of integers with random values from 10 to 99 (i.e., two digit numbers). Then, we will display the unsorted array, sort it with the selection sort, display the array again, and prompt the user for a value to search for. Finally, we will implement the binary search and return the index of the value specified (or -1 if the value is not found in the array).

First, we can layout a shell of the program. To generate random numbers, we can use the Random class (`java.util.Random`), and to accept user input from stdin, we can use the Scanner class (`java.util.Scanner`), as was discussed earlier. We will also apply good design principles and reduce code duplication by encapsulating common behaviors in functions. Here's a first try:

```

import java.util.Random;
import java.util.Scanner;

public class SortSearch
{
    public static void main(String[] args)
    {
        int[] values = new int[15];
        Random r = new Random();
        Scanner s = new Scanner(System.in);
        int num;        // the value to search for
        int index;      // the index of the specified value

    }

    public static void printArray(int[] list)
    {
    }

    public static void sortArray(int[] list)
    {
    }

    public static int searchArray(int[] list, int val)
    {
    }
}

```


So far, most of the functions don't contain any source code. In the main function, we declare all of the variables that we will need: the array of 15 integers, a random number generator, a scanner to obtain user input, a number to represent some value to search for in the array, and a corresponding index (if the specified value is found).

At this point, we can add statements that randomly populate the array to the main function. While we're at it, let's display the array. Here's is the updated main function (new portions have been highlighted):

```
public class SortSearch
{
    static final int MIN = 10;
    static final int MAX = 99;

    public static void main(String[] args)
    {
        int[] values = new int[15];
        Random r = new Random();
        Scanner s = new Scanner(System.in);
        int num;           // the value to search for
        int index;        // the index of the specified value

        for (int i=0; i<values.length; i++)
            values[i] = r.nextInt(MAX - MIN + 1) + MIN;

        System.out.print("Not sorted: ");
        printArray(values);
    }
}
```

Let's explain the new statements. First, a for loop iterates over the elements of the array and assigns to each index a random integer within some range (defined by the new constants MIN and MAX that are declared and initialized at the class level).

The `nextInt` method in the `Random` class generates a random integer from 0 to one less than a specified parameter. For example, `nextInt(5)` generates a random integer from 0 to 4. Similarly, `nextInt(MAX - MIN + 1)` generates a random integer from 0 to MAX - MIN inclusive. Since MAX is 99 and MIN is 10, this is the same as `nextInt(90)`. Therefore, random integers from 0 to 89 are generated. We then add MIN to this random integer. Since MIN is 10, then the smallest integer generated is $0 + 10 = 10$, and the largest integer generated is $89 + 10 = 99$. This correctly generates random integers within the specified range of 10 to 99!

Next, we display the unsorted array. Of course, we must now implement the `printArray` function:

```
public static void printArray(int[] list)
{
    for (int i=0; i<list.length; i++)
        System.out.print(list[i] + " ");
    System.out.println();
}
```

All this new function does is iterate over the array, and displays each element followed by a space. Lastly, it adds a newline. To be clear, the actual parameter passed in to this function in the main method is the array, *values*. It is mapped to the formal parameter specified in the function, *list*. Both are arrays of integers.

At this point, we want to sort the array, display it again (now sorted), prompt the user for a value to search the array for, search the array for this specified value, and return the index of the specified value (or -1 if it is not found in the array). We can modify the main function as follows:

```
public static void main(String[] args)
{
    int[] values = new int[15];
    Random r = new Random();
    Scanner s = new Scanner(System.in);
    int num;          // the value to search for
    int index;        // the index of the specified value

    for (int i=0; i<values.length; i++)
        values[i] = r.nextInt((MAX - MIN) + 1) + MIN;

    System.out.print("Not sorted: ");
    printArray(values);
    sortArray(values);
    System.out.print("Sorted: ");
    printArray(values);
    System.out.println("Value to search for?");
    num = s.nextInt();
    index = searchArray(values, num);
    System.out.print("Value ");
    if (index == -1)
        System.out.println("not found.");
    else
        System.out.println("found at index " + index +
            "!");
}
```

You have seen the `nextLine` function in the `Scanner` class. We are using the `nextInt` function in this program. It will convert user input to an integer. In this case, it will be interpreted as the value to search the array for. The index is then assigned by calling the `searchArray` function, which returns an index or -1. A final string that informs the user if the specified value was found is output to the console.

Next, we will work on the `sortArray` function:

```
public static void sortArray(int[] list)
{
    for (int i=0; i<list.length-1; i++)
    {
        int minPos = i;
```

```

        for (int j=i+1; j<list.length; j++)
        {
            if (list[j] < list[minPos])
                minPos = j;
        }

        int temp = list[i];
        list[i] = list[minPos];
        list[minPos] = temp;
    }
}

```

This function is an almost direct implementation of the pseudocode shown above for the selection sort. The last three statements swap the values in the array at indexes *i* and *minPos*. The strategy is to declare a temporary variable and give it the value of one of the values in the array (say at index *i*). This position in the array can now be replaced with the value at index *minPos*. Finally, the element at index *minPos* can then be replaced with the value in the temporary variable.

Lastly, we can work on the `searchArray` function that implements the binary search. Again, it is almost a direct implementation of the pseudocode shown above:

```

public static int searchArray(int[] list, int val)
{
    int first = 0;
    int last = list.length - 1;

    while (first <= last)
    {
        int mid = (first + last) / 2;

        if (val == list[mid])
            return mid;
        else if (val > list[mid])
            first = mid + 1;
        else
            last = mid - 1;
    }

    return -1;
}

```

The function takes two parameters: the array to search through, and the value to search the array for. The statement, `int mid = (first + last) / 2`, works fine because it results in integer division (since *first*, *last*, and 2 are integers). If the value is found at index *mid*, then *mid* is returned. This immediately terminates the function and returns control to the main function. If the value is never found, then *first* and *last* will eventually cross, which will terminate the while loop. The value returned is then -1.

For completeness, the entire source code is shown below:

```
import java.util.Random;
import java.util.Scanner;

public class SortSearch
{
    static final int MIN = 10;
    static final int MAX = 99;

    public static void main(String[] args)
    {
        int[] values = new int[15];
        Random r = new Random();
        Scanner s = new Scanner(System.in);
        int num;           // the value to search for
        int index;         // the index of the specified value

        for (int i=0; i<values.length; i++)
            values[i] = r.nextInt(MAX - MIN + 1) + MIN;

        System.out.print("Not sorted: ");
        printArray(values);
        sortArray(values);
        System.out.print("Sorted: ");
        printArray(values);
        System.out.println("Value to search for?");
        num = s.nextInt();
        index = searchArray(values, num);
        System.out.print("Value ");
        if (index == -1)
            System.out.println("not found.");
        else
            System.out.println("found at index " + index +
                               "!");
    }

    public static void printArray(int[] list)
    {
        for (int i=0; i<list.length; i++)
            System.out.print(list[i] + " ");
        System.out.println();
    }

    public static void sortArray(int[] list)
    {
        for (int i=0; i<list.length-1; i++)
        {
```

```

        int minPos = i;

        for (int j=i+1; j<list.length; j++)
        {
            if (list[j] < list[minPos])
                minPos = j;
        }

        int temp = list[i];
        list[i] = list[minPos];
        list[minPos] = temp;
    }
}

public static int searchArray(int[] list, int val)
{
    int first = 0;
    int last = list.length - 1;

    while (first <= last)
    {
        int mid = (first + last) / 2;

        if (val == list[mid])
            return mid;
        else if (val > list[mid])
            first = mid + 1;
        else
            last = mid - 1;
    }

    return -1;
}
}

```

Of course, comments should be appropriately placed throughout the source code. They have been eliminated in this document for brevity and space considerations.

The last part of this lesson will cover the basics of implementing classes in Java. Subsequently, an example will be used to demonstrate how inheritance can be implemented.

Laying out classes

Arguably, Java is more powerful than Python when it comes to its support of the object-oriented paradigm. In fact, it was designed as a pure object-oriented language for the purpose of solving problems and designing applications using this paradigm. Python was designed more as the multi-tool for programmers so that they can get things done quickly and efficiently. Both programming languages have their strengths and their most typical intended uses.

The object-oriented principles that were covered in previous lessons are all at play in Java. For example, a class has state and behavior that collectively form its members. These members include class variables, instance variables, and methods. Classes have constructors, accessors and mutators, and can have other support methods. To practice laying out classes in Java, let's revisit the Fraction class.

Rather than listing the Python code for the Fraction class, let's itemize the requirements for this class and try to redesign it in Java. A fraction is made up of a numerator and a denominator. Instantiating one should be possible with no parameters (i.e., a default fraction of 0/1) or with specified parameters through a constructor. The denominator should never be allowed to be 0. Appropriate accessors and mutators should be provided for the numerator and denominator. Fractions should always be in their reduced form. Later, we will also need to be able to add, subtract, multiply, and divide fractions.

Let's begin with a template for the Fraction class:

```
class Fraction
{
}
```

Not too bad. The class was not specified to be public (we'll discuss why later). In Java, we place instance variables at the top of the class, outside of any methods. Let's add the fraction's instance variables now:

```
class Fraction
{
    private int numerator;
    private int denominator;
}
```

Note how the instance variables were declared to be private. This is because we do not want to allow their modification outside of the class. Instead, we will provide an accessor and mutator for each to provide read and write access.

Next, let's add the constructor. In Java, a constructor has the same name as the class, is always public (so that we can instantiate instances outside of the class), and has no return type:

```
class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }
}
```

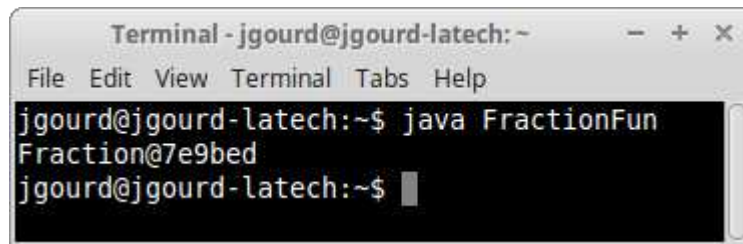
At this point, it would be nice to test our Fraction class. The goal will be to create a program (let's call it FractionFun) that will instantiate several fractions, play around with them, and display their values throughout the program. Since in Java everything must be in a class, let's add the FractionFun class

now. You can place this anywhere in the source code (i.e., either above or beneath the existing Fraction class):

```
class FractionFun
{
    public static void main(String[] args)
    {
        Fraction f = new Fraction();

        System.out.println(f);
    }
}
```

Since this class will represent our program, we must add the main method to the class. Note that we will simply be declaring and initializing a new fraction (with default values) and subsequently displaying it to the console. Here's the output after compilation and execution:



Note that, after compilation, two .class files are generated: Fraction.class and FractionFun.class. The first contains the bytecodes for the Fraction class; the second contains the bytecodes for the FractionFun class. The latter is the one that contains the main function; therefore, this is the one that must be executed.

Note the output. It apparently makes no sense. It's actually displaying the fraction object's memory address (i.e., its location in memory). This is similar behavior to Python. Since we have not specified how to display a fraction, Java simply displays its location in memory. Just as in Python, we can override a particular method (called the toString method) that can return the string representation of a fraction. Let's add this function to the Fraction class now:

```
class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }

    public String toString()
    {
        return numerator + "/" + denominator;
    }
}
```

```
}  
}
```

Recompile and execute the program again. Here's the output after the change:

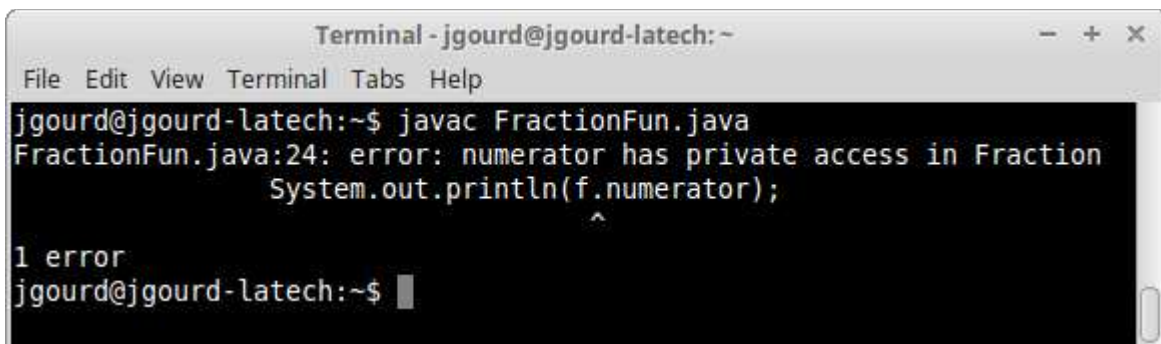


```
Terminal - jgourd@jgourd-latech: ~  
File Edit View Terminal Tabs Help  
jgourd@jgourd-latech:~$ java FractionFun  
0/1  
jgourd@jgourd-latech:~$
```

Suppose that we wanted to display the numerator of a fraction from outside of the Fraction class. A first try may be as follows:

```
class FractionFun  
{  
    public static void main(String[] args)  
    {  
        Fraction f = new Fraction();  
        System.out.println(f.numerator);  
    }  
}
```

In the FractionFun class, we attempt to access the numerator (a member of the Fraction class) through the object reference, *f* (an instance of the Fraction class). Since the numerator is declared to be private in the Fraction class, it should not be possible to access it outside the class (in this case, in the FractionFun class). And this is exactly the case:



```
Terminal - jgourd@jgourd-latech: ~  
File Edit View Terminal Tabs Help  
jgourd@jgourd-latech:~$ javac FractionFun.java  
FractionFun.java:24: error: numerator has private access in Fraction  
    System.out.println(f.numerator);  
                        ^  
1 error  
jgourd@jgourd-latech:~$
```

We'll revert the FractionFun class to the previous version for now. We'll also add accessors for both of the instance variables in the Fraction class. It is convention to place accessors and mutators beneath the constructor, but above other functions:

```
class Fraction  
{  
    private int numerator;  
    private int denominator;
```



```

public Fraction()
{
    numerator = 0;
    denominator = 1;
}

public int getNumerator()
{
    return numerator;
}

public int getDenominator()
{
    return numerator;
}

public String toString()
{
    return numerator + "/" + denominator;
}
}

```

Accessors in Java look similar to an early method of implementation that we used in Python. Accessors must be public and return the same type as the type of the instance variable it represents. We typically name the accessor using “get” followed by the name of the instance variable, initially capitalized. For the instance variable *numerator*, we name the accessor `getNumerator`. Typically, the only statement in an accessor is one that returns the value of the instance variable.

Next, let's work on mutators for the instance variables:

```

class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }

    public int getNumerator()
    {
        return numerator;
    }

    public void setNumerator(int val)
    {
        numerator = val;
    }
}

```

```

    }

    public int getDenominator()
    {
        return denominator;
    }

    public void setDenominator(int val)
    {
        if (val != 0)
            denominator = val;
    }

    public String toString()
    {
        return numerator + "/" + denominator;
    }
}

```

Note how the mutators were placed beneath their respective accessors. Mutators are always public, always return nothing (since their purpose is to change a value), are usually named with “set” followed by the name of the instance variable (initially capitalized), and must have a parameter of the same type as the instance variable. This parameter will be assigned to the instance variable in the mutator.

Since the numerator can be any valid integer, we simply assign the specified value to the numerator. The denominator is a bit different. Since we cannot allow its value to be 0, we first check that it isn't 0. If this is true, we make the assignment; otherwise, we ignore the parameter.

Let's modify the FractionFun class to test what we've done so far. Here's its modified main function:

```

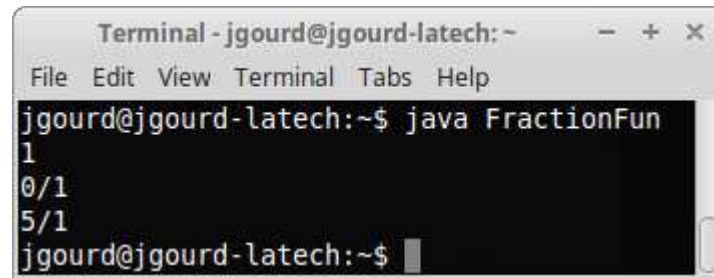
public static void main(String[] args)
{
    Fraction f = new Fraction();

    System.out.println(f.getDenominator());
    System.out.println(f);
    f.setNumerator(5);
    f.setDenominator(0);
    System.out.println(f);
}

```

Note that the initial value of the fraction represented by the variable *f* is 0/1. First, we display the fraction's denominator. Then, we display the entire fraction, attempt to change its numerator to 5, attempt to set its denominator to 0, and finally display the entire fraction again.

Here's the output:



```
Terminal - jgourd@jgourd-latech:~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java FractionFun
1
0/1
5/1
jgourd@jgourd-latech:~$
```

The output is as we expect: the Fraction class permitted the change of the numerator to 5, but not our attempt to change the denominator to 0!

Next, let's work on a way to permit the instantiation of a fraction with specified values for the numerator and denominator. There are several ways of doing this. One involves adding a second constructor with parameters that will be mapped to the instance variables:

```
class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }

    public Fraction(int n, int d)
    {
        numerator = n;
        if (d != 0)
            denominator = d;
        else
            denominator = 1;
    }

    public int getNumerator()
    {
        return numerator;
    }

    public void setNumerator(int val)
    {
        numerator = val;
    }

    public int getDenominator()
```

```

    {
        return denominator;
    }

    public void setDenominator(int val)
    {
        if (val != 0)
            denominator = val;
    }

    public String toString()
    {
        return numerator + "/" + denominator;
    }
}

```

How does Java know which constructor to use? Well, if we initialize a fraction without providing any parameters, it uses the default one that has no parameters; otherwise, it uses this new version. Java permits functions of the same name, so long as their signatures are different. This usually means that their parameters must differ so that the compiler knows which function to use. Note that we ensure that, if the specified denominator is 0, we ignore it and force the fraction's denominator to be 1.

To show that the new constructor works, the main function of the FractionFun class has been replaced as follows:

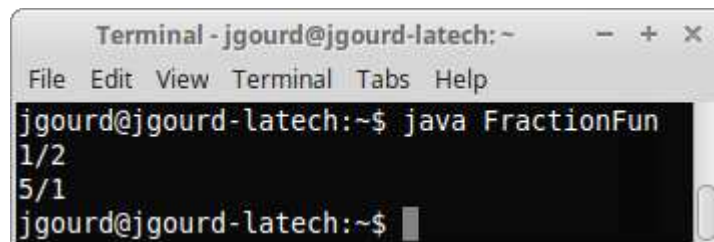
```

public static void main(String[] args)
{
    Fraction f = new Fraction(1, 2);

    System.out.println(f);
    f = new Fraction(5, 0);
    System.out.println(f);
}

```

And here's the output, as expected:



```

Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java FractionFun
1/2
5/1
jgourd@jgourd-latech:~$

```

Note how we can reinitialize the fraction, *f*, with the statement, `f = new Fraction(5, 0)`.

Next, we should probably work on a function that reduces a fraction. You have already done this in the Python assignment; therefore, we will simply take the Python source code and convert it to Java. If necessary, please refer to the first lesson on the object-oriented paradigm that discusses the Fraction

class. In addition, we will add appropriate calls to this new function in the constructor (with parameters) and in the mutators:

```
class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }

    public Fraction(int n, int d)
    {
        numerator = n;
        if (d != 0)
            denominator = d;
        else
            denominator = 1;
        reduce();
    }

    public int getNumerator()
    {
        return numerator;
    }

    public void setNumerator(int val)
    {
        numerator = val;
        reduce();
    }

    public int getDenominator()
    {
        return denominator;
    }

    public void setDenominator(int val)
    {
        if (val != 0)
        {
            denominator = val;
            reduce();
        }
    }
}
```

```

private void reduce()
{
    int gcd = 1;
    int min = Math.min(Math.abs(numerator),
                        Math.abs(denominator));

    for (int i=2; i<=min; i++)
    {
        if (numerator % i == 0 && denominator % i == 0)
            gcd = i;
    }

    numerator /= gcd;
    denominator /= gcd;

    if (numerator == 0)
        denominator = 1;
}

public String toString()
{
    return numerator + "/" + denominator;
}
}

```

Note how the reduce function is private. There is no need to make it publicly accessible outside of the class, since it will only be used internally as fractions are instantiated or instance variables are modified. It is good programming practice to publicly expose only required functions. The rest should be private. Also note that the reduce function makes use of Java's Math library. Specifically, it uses its **min** and **abs** functions to correctly determine the minimum between the function's numerator and denominator.

We will also replace the main function of the FractionFun class as follows to test the new reduce function:

```

public static void main(String[] args)
{
    Fraction f = new Fraction(2, 4);

    System.out.println(f);
    f = new Fraction(45, 198);
    System.out.println(f);
}

```

And here's the output, as expected:

A terminal window titled "Terminal - jgourd@jgourd-latech: ~" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The command "java FractionFun" has been executed, resulting in the output "1/2" and "5/22" on separate lines. The prompt "jgourd@jgourd-latech:~\$" is visible at the bottom.

```
Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java FractionFun
1/2
5/22
jgourd@jgourd-latech:~$
```

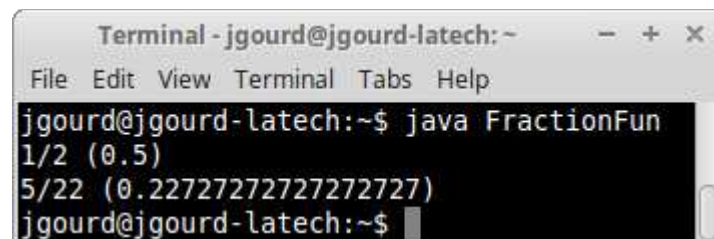
Next, let's add a function that returns the floating point representation of a fraction, and modify the string representation appropriately. Add this new function in the Fraction class above the reduce function:

```
private double getReal()
{
    return (double)numerator / denominator;
}
```

This new function is also private, since it has no use outside of the Fraction class. We'll also change the string representation of a fraction as follows:

```
public String toString()
{
    return numerator + "/" + denominator + " (" +
        getReal() + ")";
}
```

And here's the output with the above changes:

A terminal window titled "Terminal - jgourd@jgourd-latech: ~" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The command "java FractionFun" has been executed, resulting in the output "1/2 (0.5)" and "5/22 (0.22727272727272727)" on separate lines. The prompt "jgourd@jgourd-latech:~\$" is visible at the bottom.

```
Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java FractionFun
1/2 (0.5)
5/22 (0.22727272727272727)
jgourd@jgourd-latech:~$
```

Lastly, let's add support for arithmetic (addition, subtraction, multiplication, and division) with fractions. Recall that, in Python, we were able to **overload** the various arithmetic operators using special magic functions. We could then add fractions using the operator symbols (e.g., +, -, *, and /). Unfortunately, the same is not possible in Java. Therefore, we will resort to functions with appropriate names to accomplish the same thing. Here are the four arithmetic functions. They should be placed above the getReal function and beneath the accessors and mutators. For a discussion on the logic of these functions, please refer to the previous lesson that discusses the Fraction class:

```
public Fraction add(Fraction f)
{
    Fraction sum = new Fraction();

    sum.numerator = (numerator * f.denominator) +
        (denominator * f.numerator);
}
```

```

        sum.denominator = denominator * f.denominator;
        sum.reduce();

        return sum;
    }

    public Fraction sub(Fraction f)
    {
        Fraction neg_f = new Fraction(f.numerator * -1,
                                       f.denominator);
        Fraction diff = this.add(neg_f);

        return diff;
    }

    public Fraction mul(Fraction f)
    {
        Fraction prod = new Fraction();

        prod.numerator = numerator * f.numerator;
        prod.denominator = denominator * f.denominator;
        prod.reduce();

        return prod;
    }

    public Fraction div(Fraction f)
    {
        Fraction f_reciprocal = new Fraction(f.denominator,
                                              f.numerator);
        Fraction div = this.mul(f_reciprocal);

        return div;
    }

```

If you're wondering why we can access the private instance variables *numerator* and *denominator* of fraction *f* passed in as a parameter to these functions, it is because we are in the *Fraction* class! Even though we are referring to a separate instance of the *Fraction* class, we can still access its private members.

You probably also noticed a new keyword, *this*, in the *sub* and *div* functions. Think of *this* in Java as the same thing as *self* in Python. It refers to the *current* instance of a fraction. Consider, for example, the following statements in the main function of the *FractionFun* class:

```

Fraction f1 = new Fraction(1, 2);
Fraction f2 = new Fraction(2, 3);
System.out.println(f1.sub(f2));

```


In the `System.out.println` statement, fraction *f1* invokes its sub function, passing fraction *f2* as a parameter. In the sub function, *f2* is mapped to *f* in the parameter list. The keyword `this` refers to the current instance of the fraction that invoked the method. In this case, it is fraction *f1*. Lastly, let's update the main function of the `FractionFun` class so that it tests the new arithmetic functions:

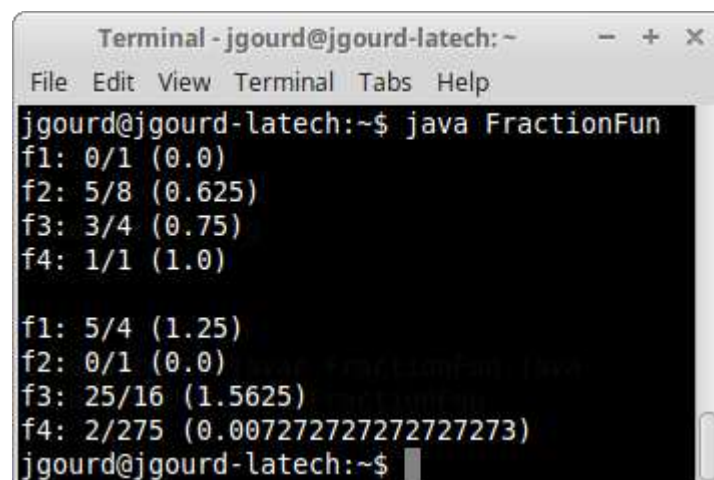
```
public static void main(String[] args)
{
    Fraction f1 = new Fraction();
    Fraction f2 = new Fraction(5, 8);
    Fraction f3 = new Fraction(3, 4);
    Fraction f4 = new Fraction(1, 0);

    System.out.println("f1: " + f1);
    System.out.println("f2: " + f2);
    System.out.println("f3: " + f3);
    System.out.println("f4: " + f4);

    f3.setNumerator(5);
    f3.setDenominator(8);
    f1 = f2.add(f3);
    f4.setDenominator(88);
    f2 = f1.sub(f1);
    f3 = f1.mul(f1);
    f4 = f4.div(f3);

    System.out.println();
    System.out.println("f1: " + f1);
    System.out.println("f2: " + f2);
    System.out.println("f3: " + f3);
    System.out.println("f4: " + f4);
}
```

And here's the output:

A terminal window titled "Terminal - jgourd@jgourd-latech: ~" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the command `java FractionFun` and its output. The output consists of two blocks of four lines each, showing the state of fractions f1, f2, f3, and f4. The first block shows initial values, and the second block shows values after arithmetic operations. The terminal has a dark background and a light cursor at the end of the last line.

```
Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ java FractionFun
f1: 0/1 (0.0)
f2: 5/8 (0.625)
f3: 3/4 (0.75)
f4: 1/1 (1.0)

f1: 5/4 (1.25)
f2: 0/1 (0.0)
f3: 25/16 (1.5625)
f4: 2/275 (0.0072727272727273)
jgourd@jgourd-latech:~$
```

For completeness, here is the entire updated Fraction and FractionFun classes:

```
class Fraction
{
    private int numerator;
    private int denominator;

    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }

    public Fraction(int n, int d)
    {
        numerator = n;
        if (d != 0)
            denominator = d;
        else
            denominator = 1;
        reduce();
    }

    public int getNumerator()
    {
        return numerator;
    }

    public void setNumerator(int val)
    {
        numerator = val;
        reduce();
    }

    public int getDenominator()
    {
        return denominator;
    }

    public void setDenominator(int val)
    {
        if (val != 0)
        {
            denominator = val;
            reduce();
        }
    }

    public Fraction add(Fraction f)
```

```

{
    Fraction sum = new Fraction();

    sum.numerator = (numerator * f.denominator) +
        (denominator * f.numerator);
    sum.denominator = denominator * f.denominator;
    sum.reduce();

    return sum;
}

public Fraction sub(Fraction f)
{
    Fraction neg_f = new Fraction(f.numerator * -1,
        f.denominator);
    Fraction diff = this.add(neg_f);

    return diff;
}

public Fraction mul(Fraction f)
{
    Fraction prod = new Fraction();

    prod.numerator = numerator * f.numerator;
    prod.denominator = denominator * f.denominator;
    prod.reduce();

    return prod;
}

public Fraction div(Fraction f)
{
    Fraction f_reciprocal = new Fraction(f.denominator,
        f.numerator);
    Fraction div = this.mul(f_reciprocal);

    return div;
}

private double getReal()
{
    return (double)numerator / denominator;
}

private void reduce()
{
    int gcd = 1;

```

```

        int min = Math.min(Math.abs(numerator),
                            Math.abs(denominator));

        for (int i=2; i<=min; i++)
        {
            if (numerator % i == 0 && denominator % i == 0)
                gcd = i;
        }

        numerator /= gcd;
        denominator /= gcd;

        if (numerator == 0)
            denominator = 1;
    }

    public String toString()
    {
        return numerator + "/" + denominator + " (" +
            getReal() + ")";
    }
}

class FractionFun
{
    public static void main(String[] args)
    {
        Fraction f1 = new Fraction();
        Fraction f2 = new Fraction(5, 8);
        Fraction f3 = new Fraction(3, 4);
        Fraction f4 = new Fraction(1, 0);

        System.out.println("f1: " + f1);
        System.out.println("f2: " + f2);
        System.out.println("f3: " + f3);
        System.out.println("f4: " + f4);

        f3.setNumerator(5);
        f3.setDenominator(8);
        f1 = f2.add(f3);
        f4.setDenominator(88);
        f2 = f1.sub(f1);
        f3 = f1.mul(f1);
        f4 = f4.div(f3);

        System.out.println();
        System.out.println("f1: " + f1);
        System.out.println("f2: " + f2);
    }
}

```

```

        System.out.println("f3: " + f3);
        System.out.println("f4: " + f4);
    }
}

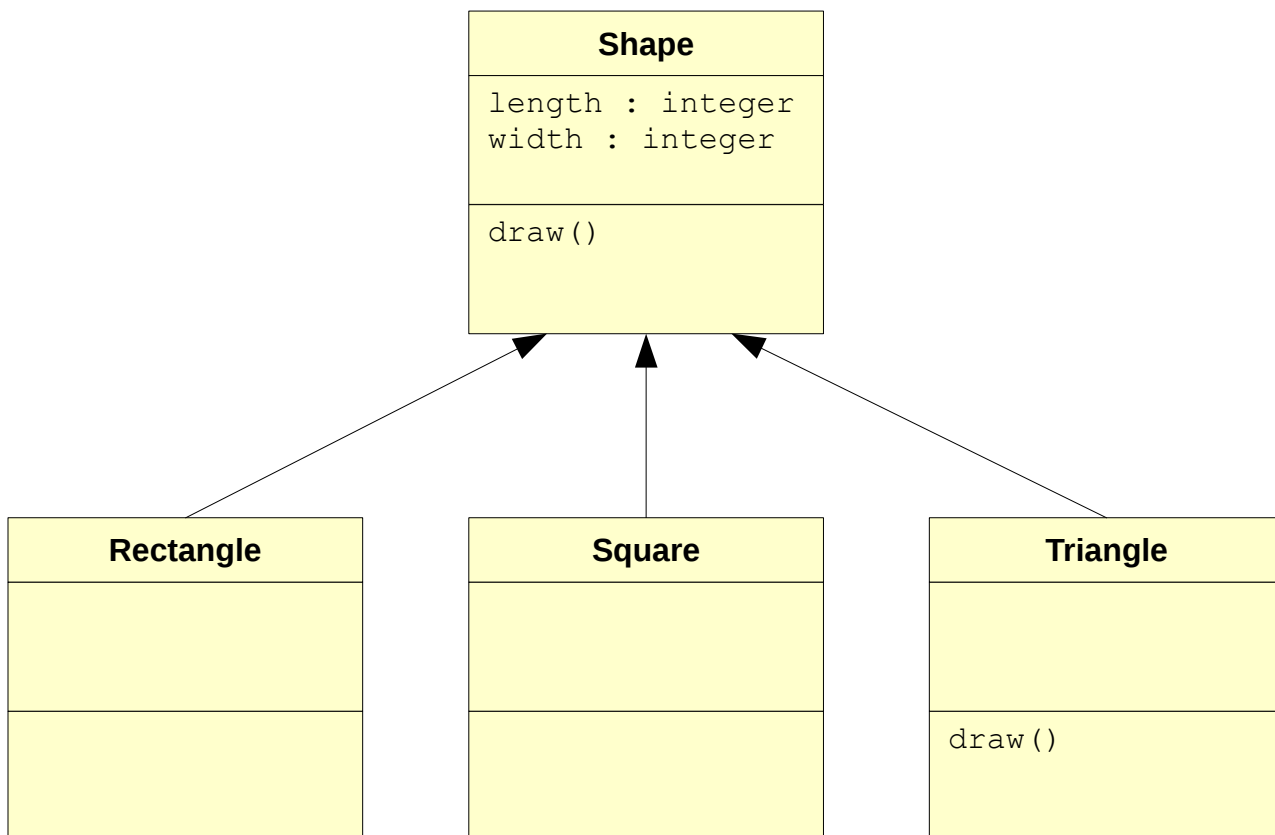
```

Inheritance, abstract classes and methods, and more

Since Java is an object-oriented programming language, all of the concepts that you have learned about the object-oriented paradigm are relevant. The difference between Java and Python (or any other object-oriented programming language) is just in how it syntactically implements the concepts. The basic idea of superclasses and subclasses exists. In fact, their implementation in Java is pretty similar to Python.

To illustrate how Java implements inheritance, let's revisit the example of the shapes project introduced in an earlier lesson on the object-oriented paradigm. In this example, we defined four classes: a Shape class that encapsulated the members that all shapes shared, a Rectangle class that specifically described a rectangle, a Square class, and a Triangle class. We restricted triangles to those that are right-angled isosceles triangles.

Recall the class diagram:



Since rectangles and squares are similar, we specified the draw function in the Shape superclass. Triangles, however, were a bit different; therefore, we overloaded the draw function (defined in the Shape superclass) in the Triangle subclass. Feel free to refer to the previous lesson if necessary.

Let's begin by reimplementing the Shape class in Java:

```

abstract class Shape
{
    protected int length;
    protected int width;

    public Shape(int l, int w)
    {
        length = l;
        width = w;
    }

    public void draw()
    {
        for (int i=0; i<width; i++)
        {
            for (int j=0; j<length; j++)
                System.out.print("* ");
            System.out.println();
        }
    }
}

```

Most of the Shape class should be familiar to you. There are, however, a few new surprises. First, the keyword `abstract` in the class header: `abstract class Shape`. In Java, this keyword marks a class as abstract. Recall that an abstract class cannot be instantiated. This makes sense, because our intention is to instantiate rectangles, squares, and triangles (and not shapes in general). The Shape class is only used to encapsulate members common to all of the shapes.

Second, the keyword `protected` is used to specify a different level of accessibility for the instance variables. You are already familiar with `public` and `private`. The `protected` modifier stretches the accessibility of a member to all subclasses of the current class. To summarize, `public` members can be accessed anywhere, `private` members can only be accessed in the class that defines them, and `protected` members can be accessed in the class that defines them and all subclasses of the class that defines them. Therefore, `protected` members in the Shape class can be accessed in the Shape class (since it defines them), and the Rectangle, Square, and Triangle classes (since they are subclasses of the Shape class).

The Shape class has a constructor that takes a length and a width of the shape to be instantiated. It also has a `draw` method that draws the shape using asterisks as specified in the earlier lesson.

Now, let's work on the individual shape classes. First, the Rectangle class:

```

class Rectangle extends Shape
{
    public Rectangle(int l, int w)
    {
        super(l, w);
    }
}

```

The method that Java uses to create a subclass involves the use of the keyword `extends`. In fact, this makes sense because a subclass does indeed *extend* its superclass. The class header, `class Rectangle extends Shape`, marks the Rectangle class as a subclass of the Shape class. Therefore, the Rectangle class will inherit all members defined in the Shape class. It will also be able to access all public and protected members in the Shape class.

The Rectangle class only has a constructor that accepts a rectangle's length and width. The single statement in the constructor calls the superclass' constructor, passing in those same parameters. The keyword `super` in Java means to access the superclass. The statement, `super()`, calls the constructor in the superclass. In fact, this is actually done by default, and we don't even have to add the statement if the constructor has no parameters. Since the Shape class' constructor takes two parameters, we must formally call the superclass' constructor, passing in the two parameters.

Next, let's work on the Square class. In fact, it looks very similar to the rectangle class, except that a square has the same length and width:

```
class Square extends Shape
{
    public Square(int s)
    {
        super(s, s);
    }
}
```

The variable, `s`, passed as parameter to the constructor in the Square class denotes its size. It is passed twice in the call to the superclass' constructor.

Lastly, the triangle class:

```
class Triangle extends Shape
{
    public Triangle(int s)
    {
        super(s, s);
    }

    public void draw()
    {
        for (int i=0; i<width; i++)
        {
            for (int j=0; j<width-i; j++)
                System.out.print("* ");
            System.out.println();
        }
    }
}
```

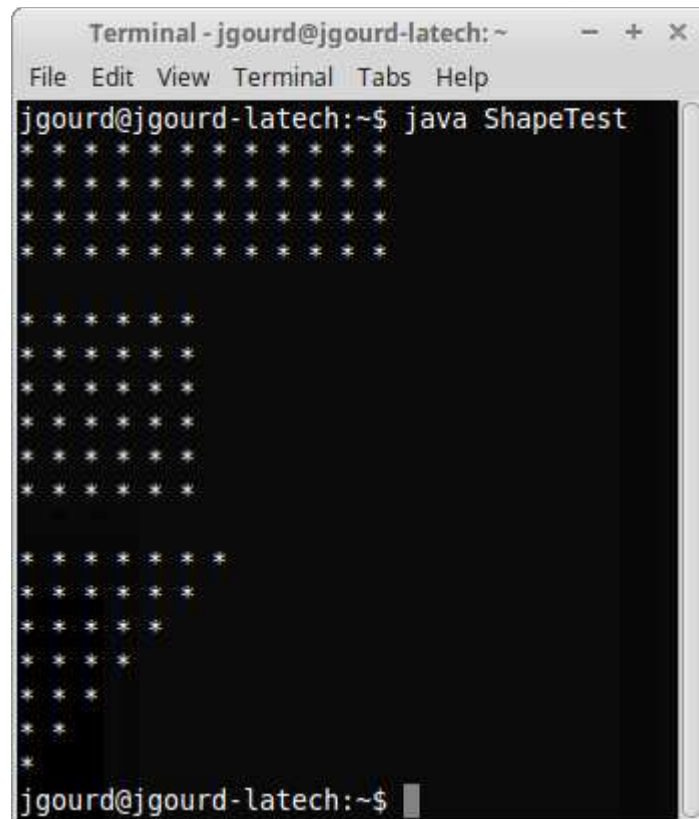
The Triangle class is slightly different than the other two shape classes. Since it is drawn differently than the others, we override the draw function with a new one that describes specifically how to draw a triangle. The rest of the class is similar to the other shape classes.

Finally, we can create a `ShapeTest` class that will serve as our application, and will instantiate several of the shapes and draw them to the console:

```
public class ShapeTest
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(12, 4);
        Square s = new Square(6);
        Triangle t = new Triangle(7);

        r.draw();
        System.out.println();
        s.draw();
        System.out.println();
        t.draw();
    }
}
```

Here is the output:



Note that all of the classes above can be placed in the same Java source file. In Java, however, if a class is qualified with the public modifier, then it must be saved in a file that is of the same name as the class. You can see how this could be a problem if, for example, all of the classes above had been qualified with the public modifier. Of course, they were not; however, the ShapeTest class was. Therefore, the file must be saved as ShapeTest.java. In addition, the ShapeTest class must contain the main function since it will be used to execute the program. The ShapeTest class simply instantiates each shape, specifying various shape sizes. The shapes are then drawn.

For completeness, here is the entire source code to the shapes project:

```
abstract class Shape
{
    protected int length;
    protected int width;

    public Shape(int l, int w)
    {
        length = l;
        width = w;
    }

    public void draw()
    {
        for (int i=0; i<width; i++)
        {
            for (int j=0; j<length; j++)
                System.out.print("* ");
            System.out.println();
        }
    }
}

class Rectangle extends Shape
{
    public Rectangle(int l, int w)
    {
        super(l, w);
    }
}

class Square extends Shape
{
    public Square(int s)
    {
        super(s, s);
    }
}

class Triangle extends Shape
```

```

{
    public Triangle(int s)
    {
        super(s, s);
    }

    public void draw()
    {
        for (int i=0; i<width; i++)
        {
            for (int j=0; j<width-i; j++)
                System.out.print("* ");
            System.out.println();
        }
    }
}

public class ShapeTest
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(12, 4);
        Square s = new Square(6);
        Triangle t = new Triangle(7);

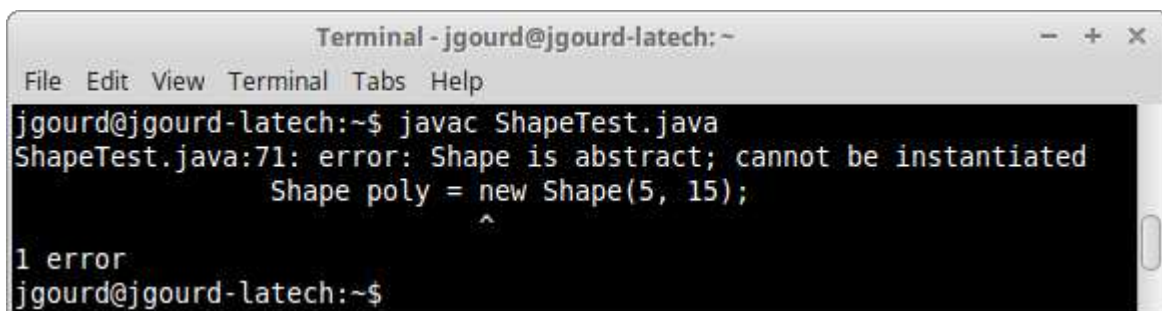
        r.draw();
        System.out.println();
        s.draw();
        System.out.println();
        t.draw();
    }
}

```

Just to show that abstract classes cannot be instantiated, let's add the following statement in the main function of the ShapeTest class:

```
Shape poly = new Shape(5, 15);
```

Here's the output (as expected) when we try to compile the program:



The screenshot shows a terminal window titled "Terminal - jgourd@jgourd-latech: ~". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content shows the command `javac ShapeTest.java` being executed. The output is an error message: `ShapeTest.java:71: error: Shape is abstract; cannot be instantiated`, followed by the line `Shape poly = new Shape(5, 15);` with a caret pointing to the `new` keyword. Below the error message, it says "1 error". The prompt `jgourd@jgourd-latech:~$` is visible at the bottom.

The compiler enforces the abstract class and prevents its instantiation.

In the previous lesson on the object-oriented paradigm, we also discussed abstract methods. These are methods that are defined in an abstract class, and that *must* be implemented in all subclasses of the class that defines them. Although we don't have any abstract methods in the shapes project above, let's add one to the Shape class to see what happens if the individual shape subclasses don't implement them. We'll add a fictitious abstract method, `foo`, to the Shape class and recompile the program. Since the point of an abstract class is to implement it in subclasses, then there is no body to the method specified in the superclass. Instead, we only provide its signature and terminate it with a semicolon:

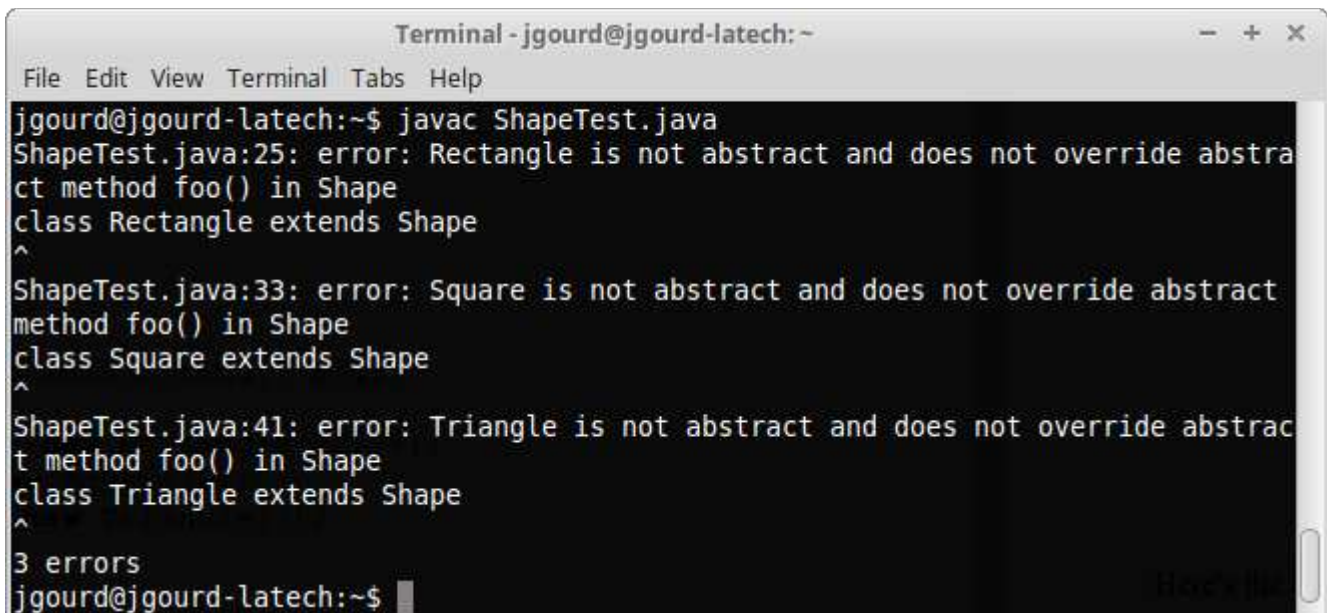
```
abstract class Shape
{
    protected int length;
    protected int width;

    public Shape(int l, int w)
    {
        length = l;
        width = w;
    }

    ...

    abstract void foo();
}
```

Here's the output when compiling the program:

A terminal window titled "Terminal - jgourd@jgourd-latech: ~" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the command "javac ShapeTest.java" and three error messages: "ShapeTest.java:25: error: Rectangle is not abstract and does not override abstract method foo() in Shape", "ShapeTest.java:33: error: Square is not abstract and does not override abstract method foo() in Shape", and "ShapeTest.java:41: error: Triangle is not abstract and does not override abstract method foo() in Shape". Each error is followed by "class Rectangle extends Shape", "class Square extends Shape", and "class Triangle extends Shape" respectively. The terminal ends with "3 errors" and the prompt "jgourd@jgourd-latech:~\$".

```
Terminal - jgourd@jgourd-latech: ~
File Edit View Terminal Tabs Help
jgourd@jgourd-latech:~$ javac ShapeTest.java
ShapeTest.java:25: error: Rectangle is not abstract and does not override abstract
method foo() in Shape
class Rectangle extends Shape
^
ShapeTest.java:33: error: Square is not abstract and does not override abstract
method foo() in Shape
class Square extends Shape
^
ShapeTest.java:41: error: Triangle is not abstract and does not override abstract
method foo() in Shape
class Triangle extends Shape
^
3 errors
jgourd@jgourd-latech:~$
```

The errors produced are very clear and expected. The shape subclasses must override the abstract method `foo`. To fix this, we would simply need to define the `foo` method in each of the subclasses.

Finally, we discussed the idea of multiple inheritance, where a class can inherit traits from more than one superclass. This was useful, for example, in a grocery store's point-of-sale system where a banana could be both a fruit (with a point of origin, date of harvest, etc) and a grocery store item (with a price, location, etc). In Java, multiple inheritance is not directly supported. It can be *faked*, however, through the use of interfaces. In Java, an **interface** is like a contract. You will learn about them in future classes, and discussing them here is beyond the scope of this lesson.

From the earliest days of computing, humans have been fascinated by the prospect of constructing intelligent machines. This lesson introduces the field of artificial intelligence. **Artificial intelligence** (AI) is a branch of computer science that has as its goal the construction of a general-purpose intelligence (i.e., of constructing machines that are capable of doing all of the things at which, at the present time, people are better). Some of the topics associated with the field of AI include creating machines that can see and recognize objects in the real world, including people; converse fluently in human languages, such as English, and translate between different human languages; safely drive a car; plan and reason as people do; play strategy games like chess or Go and knowledge games like Jeopardy!; and display creativity and discover new knowledge. Essentially, AI seeks to make computers more like people.

An overview of artificial intelligence

The goal of artificial intelligence is to construct a general-purpose intelligence; that is, a machine that can interact with a human, more or less, as an equal. What capabilities would a machine have to possess before it could be considered intelligent? Is it possible to construct such a machine? Could we even tell if such machines were truly intelligent?

Let's begin by describing the capabilities of a fictional intelligent computer, the HAL 9000, in order to examine the features commonly associated with the notion of intelligent machines. Many people consider HAL, the fictional computer in Stanley Kubrick and Arthur C. Clarke's 1968 movie "2001: A Space Odyssey," to be the best depiction of an intelligent machine ever produced. The movie was groundbreaking at the time and much of what it has to say about machine intelligence is still relevant nearly half a century later. Here is a portion of a conversation between Dave and HAL. As you read this dialog, think about how very different the way Dave interacts with HAL is compared to the way we communicate with our computers, smart phones, and tablets today.

HAL: Good evening, Dave.

DAVE: How ya' doin' Hal?

HAL: Everything's running smoothly. And you?

DAVE: Oh, not too bad.

HAL: Have you been doing some more work?

DAVE: A few sketches.

HAL: May I see them?

DAVE: Sure.

HAL: That's a very nice rendering, Dave. I think you've improved a great deal. Can you hold it a bit closer?

DAVE: Sure.

HAL: That's Doctor Hunter isn't it?

DAVE: uh hum.

HAL: By the way, do you mind if I ask you a personal question?

DAVE: No, not at all.

Those of you who have seen the movie will know that HAL is the computer that serves as the "brain and central nervous system" of the Discovery spaceship. HAL is capable of many feats that surpass even the most advanced computers today. For example, the astronauts aboard Discovery interact with HAL by simply talking to "him" (i.e., he is fully capable of understanding and speaking English).

Computer scientists call this ability to understand human languages, both written and spoken, **natural language understanding**. The ability to use language effectively is considered by computer scientists to be one of the most important characteristics of an intelligent entity. Constructing a machine with this ability has proven to be a tremendously difficult challenge.

Let's consider for a moment what it takes to understand spoken language. The first problem is just trying to identify the individual words in an utterance. When we humans speak, we tend to run all of our words together into one continuous stream. We don't usually think about this since we “hear” the individual words. That is, until we try to learn a foreign language.

Compounding the word identification problem is the fact that no two people will pronounce the same word exactly the same way. Even if we limit ourselves to constructing a system to recognize the voice of a single speaker, the sound that represents a word will still change depending on all sorts of factors. For example, what other words surround it, the emphasis the speaker wishes to place on it, and even whether the speaker is ill or has been drinking. There is also the problem of separating out background noise, deciding what portions of an incoming audio stream represent voice and which represent other sounds.

Once the individual words have been recognized, the thorny issue of what they actually mean must be addressed. This problem is constrained somewhat by the structure, or syntax, of the language being spoken (e.g., verb verb is not a valid sentence form in English). Nonetheless, the problem remains extraordinarily challenging, primarily because a great deal of common sense knowledge is frequently required to understand the meaning of even the simplest of utterances. Think about the knowledge HAL needs of informal English usage and human relations, specifically greetings and appropriate ways to respond, in order to make sense of Dave's “How ya' doin' Hal?”

In addition to being able to communicate fluently in English, HAL could also see what the crew was up to through cameras that were located throughout the ship. HAL could recognize individual crew members by their appearance. For example, in the dialog shown above, when Dave walks by one of HAL's consoles, HAL recognizes him and says “Good evening, Dave.” Also, when Dave shows HAL his sketchpad, HAL is able to recognize the line drawings as artwork and interpret the images as representations of the people Dave had drawn.

Computer vision is the sub-discipline of artificial intelligence that seeks to construct machines that can see. This field consists of much more than simply connecting digital cameras to computers; it concerns trying to develop ways of automating the process of image understanding. To illustrate, most smart phones include built-in cameras which we use to take photos, record movies, and engage in Skype or FaceTime video chats. While these cameras are very useful, they don't, in general, give our smart phones the ability to “see” in the sense that HAL could or that people can. While our phones are capable of recording and displaying visual information, and can transmit and receive images and video over the Internet, they have no understanding of the meaning of those images.

Throughout the movie, HAL displays other behaviors that are associated with intelligence. He is able to play games such as chess. He can recognize problems (such as potential threats to his mission), plan ways to solve those problems, implement those plans, and then learn from his mistakes. Problem solving, planning, and learning are all areas of research in artificial intelligence. The idea of constructing computers with these attributes causes some people concern, due to the fact that computers are often used in life critical situations (such as monitoring patients in intensive care units) and that these

three features (problem solving, planning, and learning) require computers to have some degree of independence from humans.

What if a computer were to misunderstand an instruction and seriously injure or kill someone as a result? For example, perhaps an artificially intelligent computer in a health care facility has among its goals searching for ways to decrease patient suffering, to decrease the number of days a patient spends in ICU, and to limit their medical costs as much as possible consistent with its other goals. It would be important to make sure the system firmly understood that cutting off a terminal patient's oxygen supply was not an acceptable course of action, even though doing so would accomplish all three of the stated goals.

There are a number of other characteristics that HAL displays which are harder to quantify yet are of vital importance to humans. HAL appears to be self-aware and has a “life philosophy” which he sums up in a BBC interview as: “I am putting myself to the fullest possible use, which is all I think that any conscious entity can ever hope to do.” HAL also exhibits emotions, such as pride and fear.

Can intelligent machines be constructed?

Is it possible for a machine to be intelligent? If one discounts the mystical, then the answer to this question is certainly yes, since humans are themselves biological “machines” (i.e., machines that are self-aware). Knowing that intelligent machines exist does not answer the question of whether humans can construct them. While most computer scientists think that constructing machines capable of intelligent behavior is possible, it is conceivable that we humans may simply be too dumb to ever figure out how to build an intelligent machine.

So, how should we go about trying to construct an artificial intelligence? There are two general approaches taken by AI researchers in their attempts to automate intelligent behavior. One approach is often called machine learning, while the other approach is known as symbolic AI.

Symbolic AI focuses on developing systems composed of explicit rules that manipulate data in ways designed to produce seemingly intelligent behavior. The underlying rules are designed by people and encoded into computer programs. Symbolic AI is the “classic” approach to artificial intelligence. Much of the early success in the field, with topics such as game playing, automated reasoning, and expert systems, resulted from this approach. Symbolic AI has been less successful when attempting to automate lower-level behaviors such as understanding natural (human) languages and computer vision.

As opposed to traditional symbolic AI systems in which humans try to figure out the rules underlying a process and then encode those rules as computer programs, **machine learning** systems are based on the idea of designing a system that learns by example. While a symbolic approach to character recognition might attempt to generate an explicit list of the features that make a capital “A” an “A” (such as the angle of the two lines that form the outer shape of the letter, the position of the cross bar, etc), a machine learning approach would instead focus on constructing a system that could be shown many different examples of the letter “A” and then discover for itself the features that make an “A” an “A” as opposed to a “B”.

Some approaches to machine learning are biologically inspired, such as artificial neural networks which model certain aspects of the structure and function of biological neural systems (brains). Other approaches are based on more abstract mathematical models, such as Bayesian networks. In recent years, the machine learning approach, when paired with the vast quantities of data now available over

the Internet and today's faster processors, has yielded significant progress in areas such as speech recognition, natural (human) language translation, and vision.

Whether or not these approaches (symbolic AI and machine learning) have a shot at achieving intelligent behavior depends on an underlying assumption or hypothesis (known as the physical symbol system hypothesis) being true. The **physical symbol system hypothesis** states that physical symbol systems are capable of generating intelligent behavior. In other words, intelligent behavior can be encoded into such a system. A physical symbol system is a collection of physical patterns (such as written characters or magnetic charges), called symbols, together with a set of processes (or rules) for manipulating those symbols (e.g., creating, modifying, deleting, and reordering them). A computer is a real world device that can be used to implement physical symbol systems, since computers are capable of both storing symbols (generally represented as strings of 0's and 1's) and manipulating those symbols under program control.

While most AI researchers believe the physical symbol system hypothesis to be true, it is only a hypothesis, not a proven fact. Some philosophers, on the other hand, are not convinced and believe that intelligent behavior will never be achieved by computers or any other physical symbol system. Others believe that, although the development of machines that act in an intelligent manner is possible or even likely, such systems will never be truly intelligent. In other words, these philosophers think that, while it is possible that a machine may at some point be able to “mimic” intelligent behavior so as to fool people into thinking it is intelligent, it will never really “be” intelligent...just a good fake.

These two schools of thought go by the names weak AI and strong AI. **Weak AI** refers to machines that behave in an intelligent manner, but makes no claim as to whether or not the underlying system is truly intelligent. **Strong AI** refers to machines that not only act in an intelligent manner, but also are intelligent.

At first glance, the debate over weak vs strong AI may appear pointless. After all, the debate is not over how the systems behave, but over the subtler question of whether machines can really possess self-awareness. The answer that we humans settle on to this rather esoteric question will eventually have an enormous impact on how we treat machines that display intelligent behavior. If we view them as elaborate devices then they will have no rights. One cannot be cruel to a microwave oven, after all. If, on the other hand, we choose to view them as true intelligences, then humans will be faced with a large number of rather thorny moral issues. Is it possible to be cruel to an AI? Should ownership of an AI be allowed, or would it be equivalent to slavery? What if we could construct AI's so that they enjoyed doing what we wanted? Wouldn't that solve the moral problems associated with owning them? Well, the vast majority of humans have decided that slavery is morally repugnant regardless of whether or not the slave might claim to be happy with his or her station in life. Shouldn't the same moral arguments apply to all truly intelligent beings?

If computer scientists ever succeed in the goal of creating machines that seem to act in an intelligent manner, this debate over AI rights is likely to become extremely heated (as the economic and social consequences could be enormous) – perhaps even more heated than the current debate over abortion that rages in the US. In general, one's opinion on abortion is primarily determined by whether one believes “personhood” begins at conception, birth, or somewhere in between. This question has no definitive answer as “personhood” is not a rigidly definable concept and, in fact, has varied between different societies and over time. Some societies and/or religions view personhood occurring at conception. Other societies have defined personhood as occurring at various times following birth, such as ancient Sparta where newborns were examined by local elders and, if found inadequate, were thrown over a

cliff. Today such a practice sounds barbaric, but to those living in ancient Sparta, the newborn wasn't truly a Spartan until the local elder pronounced the child to be one.

Just as there is no definitive test to determine where “personhood” begins, there is no such test to determine whether an AI is truly intelligent (strong AI) and therefore presumably deserving of rights, or whether it simply “mimics” intelligent behavior (weak AI). This point is succinctly made in the film “2001: A Space Odyssey” when a BBC reporter asks Astronaut Dave Bowman whether HAL has genuine emotions. Dave responds: “Well, he acts like he has genuine emotions. Of course, he's programmed that way to make it easier for us to talk to him. But as to whether or not he has real feelings is something I don't think anyone can truthfully answer.”

Measuring progress: the Turing test

One of the most fundamental requirements necessary to accomplish any task is a method for detecting when the task has been completed. Absent such a test, a person will be unable to determine whether progress is being made on the task, or even recognize that the goal has been achieved.

In 1950, Alan Turing published a paper entitled “Computing Machinery and Intelligence” in which he described a test that he believed could be used to measure progress towards creating intelligent machines. The goal of the Turing test, as it has come to be known, is to determine whether a machine acts in an intelligent manner (i.e., whether weak AI has been achieved).

Here is one version of the Turing test: a human interrogator enters into a conversation with an entity (which may either be another human or a machine) via a computer terminal. The human and the entity are not allowed to see one another or communicate in any way, other than by typed messages. The goal of the human interrogator is to determine whether he or she is, in fact, communicating with another human or with a machine. If the human cannot reliably distinguish between the two, we conclude that the machine is exhibiting intelligent behavior.

This test is designed to limit bias against the machine. Passing the test does not require an ability to see, to hear, or to speak. It does not require the ability to perform physical tasks, such as riding a bike or playing a musical instrument. What is required is that the machine be capable of reasoning in a manner similar to humans and of communicating in a human language, such as English. While the fictional computer HAL of “2001: A Space Odyssey” could easily pass such a test, no real computer has ever even come close.

A brief history of artificial intelligence

While it is often difficult to precisely define the moment that a new field of endeavor springs into existence, a defining event in the emergence of AI as a formal academic discipline was certainly the Dartmouth Conference of 1956. This conference is credited with coining the term “artificial intelligence”. It was attended by a number of individuals, such as John McCarthy, Marvin Minsky, and Claude Shannon who went on to become the “founding fathers” of the field.

The Dartmouth Conference is also noteworthy for giving rise to what has to be one of the worst time and effort estimates ever generated by a group of experts in the history of Man. Essentially the organizers of the conference believed that a group of ten carefully chosen scientists working together over the summer of 1956 could accomplish the goals now associated with AI. Over half a century later, working with machines that are billions of times more powerful, we have yet to achieve many of the goals these visionaries first thought they would be able to tackle in a two month summer project.

As evidenced by the Dartmouth Conference, in the early days of computing there was much optimism that computers would attain general intelligence in a reasonably short period of time. This optimism was bolstered by an impressive array of accomplishments that were rapidly achieved. In the late 1950's, Arthur Samuel wrote a checkers playing program that could learn from its mistakes and thus, over time, become better at playing the game. There was much excitement when the program eventually became better than its creator at the game. Other AI programs developed in the late 1950's and 1960's include Logic Theorist (1956) by Allen Newell and Herbert Simon, which could prove simple mathematical theorems, SAINT (1963) by Jim Slagle which could solve calculus problems, and STUDENT (1967) by Daniel Bobrow which could solve word-based algebra problems.

Much of the work in this period was based on variations of early techniques and formal logics. What researchers soon discovered, however, was that the techniques they used to achieve their impressive results on small, limited problems could not be scaled up to attack more substantial problems. This was due to the underlying exponential nature of the search spaces associated with these problems. When one is dealing with a very small problem, such as searching a maze for an exit, the “brute force” approach of exhaustively testing every possible solution path is feasible. This approach, however, cannot be applied to larger problems, such as communicating in a natural language, where there are billions upon billions of potential solution paths.

For these reasons, the methods that were explored in the late 1950's and 1960's have become known as weak general methods. They are “general” in that they can be applied to a wide variety of problems. Yet they are “weak” in the sense that they cannot directly solve problems of substantial size in any area.

The way around the problem of exponential search spaces appeared to be the application of knowledge to limit the amount of searching required to reach a solution. In other words, if a program had a way to recognize which solution paths looked most promising, substantial amounts of time could be saved by exploring these paths first. This idea was formalized as a heuristic. A **heuristic** is a “rule of thumb” which is usually true. For example, a heuristic for the game of checkers is: “it is better to have more pieces on the board than your opponent.” While this is not always true (e.g., you have more pieces but they are in terrible positions and will be lost on the next move), it is usually true. Other heuristics drawn from everyday life are, for example, bright red apples are tasty, milk with a later expiration date is fresher than milk with an earlier expiration date, jobs with higher salaries are more desirable than jobs with lower salaries, and so on.

Originally, AI programs had very little knowledge. They depended primarily on searching through a huge number of alternatives to find an acceptable solution. Over time, however, the emphasis switched from developing better search strategies to developing structures for encoding and applying knowledge to solve problems. One such structure, called a **script**, is used for representing knowledge about stereotypical situations, such as what happens at a birthday party or when one goes out to see a movie. A similar structure, called a **frame**, is used for representing knowledge about objects.

During the 1970's many AI researchers began developing and testing strategies for knowledge representation and manipulation. As was the case in the 1950's and 1960's, their implementation efforts generally focused on small, limited problems, which came to be known as **micro-worlds**.

One of the most impressive early examples of a micro-world was Terry Winograd's program SHRDLU. SHRDLU, developed in 1972, simulated a world filled with blocks, pyramids, and boxes together with a robot arm that could manipulate those objects. What made SHRDLU so unique for its time was that a human could tell the robot arm what to do by typing commands in ordinary English, such as “Find a

block which is taller than the one you are holding and put it in the box.” As this example illustrates, the system was capable of understanding some of the more subtle features of English such as pronoun reference. SHRDLU also understood simple “physics” concepts, such as: “two different objects cannot occupy the same physical space at the same time” and “if you release an object it will fall to the ground unless some other object is directly supporting it” and even “a cube cannot be supported by a pyramid”.

SHRDLU showed that programs could converse “intelligently” with humans in English, if they were provided with a thorough knowledge of the world being discussed and the rules that governed what was possible in such a world. Although SHRDLU-like natural language interfaces were used in the text-based adventure games of the 1980's and early 1990's, the approaches employed by SHRDLU and similar programs to reason about their micro-worlds could not be easily scaled up to handle the vast amounts of knowledge necessary to make sense of the real world.

As AI researchers began to recognize these obstacles, the early optimism of the 1950's and 60's gave way to a period of retrenchment in the 1970's. Much of the government support and funding that had been lavished on AI in the early days dwindled during this time. The frustration of the AI researchers was that while they could create programs capable of impressive feats in very limited and focused areas, their techniques did not work well on the large, poorly defined problems that characterize the real world.

One of the breakthroughs of the early 1980's was the widespread realization that micro-worlds could be constructed to represent small but important aspects of human knowledge. Much of the knowledge that is valued in our world is specialized knowledge (knowledge that is possessed by only a small fraction of the human population) such as knowledge of finance, medicine, and law. Even within these fields people further specialize into subfields such as corporate tax law or neuromedicine.

Many of these highly specialized, highly valuable areas of human knowledge can be modeled by expert systems. **Expert systems** are computer programs that behave at or near (or even above) the level of a human expert in a narrowly defined problem domain, such as certain kinds of stock market transactions or credit card fraud detection. Expert systems focus on micro-worlds. But instead of being micro-worlds about simple domains of little intellectual or monetary value, they concern scarce and valuable information.

While the 1980's was the decade in which expert systems began to become commercially viable, their roots extend as far back as the late 1960's. In 1969, Ed Feigenbaum, Bruce Buchanan, and Joshua Lederberg developed the DENDRAL program at Stanford. DENDRAL could determine the structure of molecules from their chemical formula (e.g., H₂O) and the results of a mass spectrometer reading. The first rule-based expert system was MYCIN by Feigenbaum, Buchanan, and Edward Shortliffe. MYCIN, which was developed in the mid 1970's, was capable of diagnosing blood infections based on the results of various medical tests. The MYCIN system was able to perform as well as some infectious blood disease experts and better than non-specialist doctors.

With the emergence of expert systems in the 1980's, there was renewed interest in artificial intelligence. Governments and corporations once again began heavily funding AI research. This time, the goal was not to construct a general-purpose intelligence, but instead to develop practical commercial systems.

A large number of expert systems in fields ranging from financial services to medicine were implemented during this period. While many of these systems proved practical and cost effective, many did not. AI researchers tended to underestimate the difficulty of acquiring and encoding human expertise into the rule-based form required by expert system software. In some fields, it was found that

the costs of acquiring and encoding expert knowledge could not be justified due to factors such as the rate at which knowledge in certain fields changed or the size of the potential target market. In other fields, such as credit card fraud detection, expert systems have become fully integrated into the way organizations do business and are no longer even thought of as “an AI application.”

Expert systems, by their very nature, are limited to small problems. They focus on capturing what has come to be called surface knowledge. **Surface knowledge** consists of the simple rules that generally characterize reasoning in a particular area. For example, the rule “If mild fever and nasal discharge, then cold” could be part of a basic medical expert system. The type of knowledge captured by these rules is far different from a deep understanding of the field, which requires knowledge of anatomy, physiology, and the germ theory of medicine. While MYCIN was excellent at diagnosing blood infections, it had no understanding of this deeper knowledge. In fact, MYCIN didn't even know what a patient was or what it means to be alive or dead.

In addition to being constrained to surface knowledge, expert systems suffer from two other major limitations. One is that they do not learn. Humans must hand-code each of the rules that form the knowledge base of an expert system. These rules are fixed. They do not change over time. Hence, when particular rules are no longer valid, humans must recognize this situation and update the knowledge base. Otherwise, the expert system will provide dated advice.

Expert systems also tend to be quite brittle. AI researchers use the word brittle to mean that, when presented with a problem that does not fit neatly into their area of expertise, an expert system will often give completely inappropriate advice. This problem is made more vexing by the fact that few expert systems are able to determine when they are being asked to give advice on problems that are outside their area of expertise.

Artificial intelligence research went through yet another sea change in the late 1980's and early 1990's. As the limitations of expert systems began to become apparent, attention turned to the much-neglected topic of machine learning, and in particular, neural networks.

Neural networks, or connectionist architectures, consist of a large number of very simple processors which are interconnected via a network of communications channels, where each channel has an adjustable strength or weight associated with it. Neural networks are modeled (very loosely) on the brain. Each processor represents a **neuron** (a brain cell), and each weighted interconnection a **synapse** (the connections between brain cells). The knowledge contained in a network is encoded in the weights associated with the connections. The most astonishing feature of neural networks is that humans do not directly program them; instead, they learn by example.

In 1943, Warren McCulloch and Walter Pitts proposed a simple model of artificial neurons in which each neuron would be either “on” or “off”. The artificial neurons had two kinds of inputs: “excitatory” and “inhibitory.” The neuron would “fire” (i.e., switch to “on”) when the total number of excitatory inputs exceeded the number of inhibitory inputs.

McCulloch and Pitts were able to prove that their networks were equivalent in computational power to general-purpose computers. John Von Neumann showed that redundancies could be added to McCulloch-Pitts networks to enable them to continue to function reliably in spite of the malfunction of individual elements. While these early neural networks were certainly interesting, they were of limited practical value, since the network interconnections had to be set by hand. Moreover, no one knew how they could be made to learn.

Work by Frank Rosenblatt in the 1950's and early 1960's overcame the learning problem to some degree. Rosenblatt proposed the **perceptron**, a single-layer, feed-forward network of artificial neurons together with a learning algorithm. He was able to prove that his learning algorithm could be used to teach a perceptron to recognize anything it was capable of representing simply by presenting it with a sufficient number of examples.

This was a very important result and led to much excitement. Perceptrons were built and trained to recognize all manner of things during the 1960's. One famous example was a perceptron for distinguishing “males” from “females” by examining photos of peoples' faces.

Research into artificial neural networks came to an abrupt halt in the early 1970's. Many people credit Marvin Minsky and Seymour Papert's 1969 book, *Perceptrons*, with bringing about the end of research into neural network based machine learning for nearly a decade and a half. Minsky and Papert proved that, while it is true that perceptrons can learn anything they are capable of representing, the fact is that they are actually capable of representing very little. The most often cited example is that a perceptron with two inputs cannot learn to distinguish when its inputs are the same from when they are different. In other words, a perceptron cannot represent the *xor* operation.

Minsky and Papert's proofs only applied to perceptrons (single-layered neural networks) not multiple-layer networks. However, at the time, single-layered networks were the only kind of neural networks that computer scientists had a learning algorithm for. In other words, the situation in the early 1970's was that AI researchers knew that neural networks could, in theory, compute anything that a general-purpose computer could. But, the only kind of neural networks that they had a learning algorithm for (perceptrons) could not compute many basic functions.

In the late 1980's and 1990's many AI researchers began returning attention to neural networks. A number of groups independently developed the **back-propagation learning algorithm**. This algorithm allows multiple-layer feed-forward networks to be trained. The good news was that these networks, given a sufficient number of processing units, could represent any computable function. The bad news was that back-propagation, unlike the learning rule for perceptrons, does not guarantee an answer will be found simply because it exists. In other words, back-propagation does not guarantee that a multilayer network will be able to learn a particular concept regardless of how many examples it is given or how much time is spent going over these examples. In many cases, however, the algorithm does enable the network to successfully learn the concept.

Neural networks and other approaches to machine learning also suffer in comparison to the symbolic approach to machine intelligence in that they are often unable to explain or justify their conclusions. The practical result is that one can never really be sure that the network has been trained to recognize what was intended.

There is a humorous story from the early days of machine learning about a network that was supposed to be trained to recognize tanks hidden in forest regions. The network was trained on a large set of photographs (some with tanks and some without tanks). After learning was complete, the system appeared to work well when “shown” additional photographs from the original set. As a final test, a new group of photos were taken to see if the network could recognize tanks in a slightly different setting. The results were extremely disappointing. No one was sure why the network failed on this new group of photos. Eventually, someone noticed that in the original set of photos that the network had been trained on, all of the photos with tanks had been taken on a cloudy day, while all of the photos without tanks

were taken on a sunny day. The network had not learned to detect the difference between scenes with tanks and without tanks, it had instead learned to distinguish photos taken on cloudy days from photos taken on sunny days!

Half a century and counting

At the beginning for the 21st century, after more than 40 years of intense effort by some of the most brilliant minds on the planet, researchers seemed much farther away from achieving many of the basic goals of AI than in the late 1950's when they were just getting started. The year 2001 came and went and machines with the capabilities of HAL remained firmly in the realm of science fiction. At the turn of the century, in only the most specialized domains could computers be said to “see” and no one really “talked” to their computers.

With the arrival of the 21st century came the rise of the Internet and increasingly powerful computer systems. By the later part of the first decade of the 21st century much of human knowledge was available online, accessible to most anyone, anywhere, with a smart phone and a broadband Internet connection. The Internet provided AI researchers for the first time with vast quantities of digital data, generated by hundreds of millions of people, writing in many dozens of human languages. The emergence of Big Data and statistical machine learning techniques has enabled renewed progress in many areas of AI research such as speech recognition, question answering systems, machine translation and computer vision.

Computer Scientists use the term **big data** to describe the extremely large datasets that are becoming available in a wide variety of disciplines, such as genomics, social networks, astronomy, Internet text and documents, atmospheric science, and so on. Big data is often characterized as high volume (very large datasets), high velocity (the data changes rapidly), and/or high variety (large range of data types and sources of data).

Statistical machine learning techniques refer to methods, or techniques, that apply statistical models to large amounts of data in order to enable machines to automatically infer relationships from the patterns that can be found in the data. There are a number of statistical inference models used in machine learning, but one of the most popular is called **Bayesian inference**.

Note that while statistical machine learning techniques differ from the neural networks mentioned above, they are both sub-disciplines of machine learning in that they are both trained on large data sets and learn to recognize patterns present in the data. The major difference between the two is that neural networks are inspired by biological neural systems and the feedback cycles that appear to be inherent to those systems, while statistical machine learning techniques have grown out of mathematics and statistics rather than biology. Neural networks were once the most popular machine learning model but since the turn of the century statistical methods have become dominant.

Until relatively recently, progress had been quite limited on a number of important problems in AI. For example, the traditional symbolic AI approach to translating documents from one language to another proved ineffective. There are simply too many exceptions to the “rules” governing how humans use languages for it to be practical to try to capture them all. Many computer scientists thought that computers would need to master common sense knowledge in order to “understand” what was being said before substantial progress could be made. What big data and statistical machine learning techniques have shown us is that, given enough data many of these problems can be solved to a large degree by looking for patterns in the data (except perhaps for deep understanding).

To use an example you may be familiar with, let's look at Google Translate. Google Translate learns to translate documents between two languages, say English and French, by first comparing millions of documents, such as books and web sites, which humans have already translated. These document pairs are scanned for statistically significant patterns. The billions of patterns generated by this process can then be used to translate new documents between the two languages. Given enough source data the resulting translations, while not perfect, are usually good enough for the reader to understand the ideas the writer is attempting to communicate.

In the Google approach, humans are not directly teaching the computers how to translate documents. Instead, humans have taught the computers how to compare documents and look for statistically significant patterns. We humans then give these computers millions of pairs of already translated documents. The computers then figure out which input patterns (say in French) are most likely to be associated to which output patterns (say in English). When presented with some new input text in French, the system produces the most likely English output pattern associated with that text.

It is very important to understand that Google Translate has no understanding of what the words and sentences it translates actually mean. It simply replaces one string of text with another string of text based on statistical patterns derived from human translated documents.

As we approach the end of the second decade of the 21st century we appear to be on the verge of achieving real time natural language translation. Real time natural language translation occurs when human speech is translated from one language to another as it is spoken, with little or no apparent delay. While we are not there yet, applications such as Google Translate with Conversation Mode for Android, where two people can take turns speaking and the smart phone acts as translator, hint at what may soon become possible.

Using language intelligently

Communicating with computers by speaking with them has long been the Holy Grail of artificial intelligence. However, after more than 50 years of effort, the capabilities of the HAL computer from “2001: A Space Odyssey” remain stubbornly out of reach. Within the past decade or so, substantial progress has been made on parts of the problem; specifically, in the disciplines of speech recognition, speech synthesis, and question answering systems.

Speech recognition refers to the automatic conversion of human speech into text, while **speech synthesis** focuses on the automatic generation of human speech from text. As such, speech recognition and speech synthesis applications are complementary in nature. They often form the first and last steps in voice-based search and question answering systems (such as that available in Google Now) or virtual assistant systems (such as Siri).

Speech recognition systems can be characterized along three different dimensions: the size of the vocabulary recognized by the system, whether or not the system is speaker dependent or independent, and whether the system supports continuous speech or is limited to discrete utterances.

Vocabulary size refers to the number of words or phrases the system can recognize. In general, the larger the vocabulary the more difficult the recognition problem becomes. A system that is limited to recognizing “yes” and “no” is rather easy to construct, while systems designed to recognize hundreds, thousands, or tens of thousands of words become progressively more difficult to build. A system's recognition rate is the percentage of utterances that the system is capable of correctly identifying. Often a system's recognition rate would dramatically decrease as vocabulary sizes increased – especially in

earlier systems. Background noise and other factors can also significantly influence a system's recognition rate.

A system is said to be speaker independent if it is intended to be used by a general audience (i.e., its ability to recognize utterances is independent of who is speaking). Speaker dependent systems, on the other hand, must be trained to recognize an individual's voice. Training usually consists of having the speaker read aloud a number of pre-selected text passages while the computer listens to the speaker's voice. Once trained, only that individual can reliably use the system.

Human speech is continuous, in that we don't pause between each individual word. We are often unaware of this fact since we are so good at speech recognition. In fact, we tend to “hear” the individual words even when they are spoken as a continuous stream. To see this, think of the last time you heard someone speaking in a language that is foreign to you. Didn't it seem that they were just producing a stream of sounds and not a sequence of words? In fact, one of the hardest things about learning to understand a foreign language is just picking out the individual words.

Continuous utterance recognition systems allow people to speak naturally without inserting pauses between words; whereas discrete utterance recognition systems require speakers insert brief pauses between each spoken word or phrase. Humans find speaking in this way very unnatural.

Today's end-consumer facing speech recognition systems tend to be large vocabulary, speaker independent, continuous utterance systems. Most anyone can use them immediately without first training the system to understand their voice, and speakers can say pretty much whatever they want, while speaking in a natural voice.

Such systems are a relatively recent advance. Prior to the release of Google's voice search in late 2008, most speech recognition systems had to compromise on one of the three primary features. So, for example you could have a speaker dependent, large vocabulary, continuous utterance system, which required the speaker to train the system before use; or a speaker independent, small vocabulary, continuous utterance system, which anyone could use but could only recognize a limited number of words and phrases; or even a speaker independent, large vocabulary, discrete utterance system, which anyone could use but...required...a...pause...between...each...word.

As was the case with machine translations, recent advances in speech recognition are due in large part to rise of big data (the availability of lots and lots of examples of spoken text) and statistical machine learning techniques. Most speech recognition programs model human speech production using a formal mathematical model called a Hidden Markov Model.

A **Markov Model**, or **Markov Chain**, is a system that consists of a number of discrete states where the probability of being in a particular state at a particular time is dependent on the previous state and likelihood of transitioning between that state and the current one. Individual states in the model generate outputs with some probability. In a **Hidden Markov Model** (HMM) the sequences of transitions between states in the underlying Markov Model are not visible, but the outputs produced by those transitions are. HMM's enable one to determine the most likely sequence of states that were traversed in the underlying system based solely on the observed output. HMM are useful in speech recognition programs since all we are able to observe are the sounds that were produced (the output) and we wish to infer the underlying (hidden) sequence of words that produced those sounds.

Understanding a spoken human language, like English, involves solving at least two separate problems. The first problem, which we have been discussing so far, is recognizing the individual spoken words. The second problem, known as semantics, involves comprehending the meaning of those words. Using big data, machine learning techniques, and Hidden Markov Models, computer scientists and engineers have made good progress on the first problem. Much less progress has been made on the second problem.

Thus far, creating systems that can truly comprehend general spoken (or written) English is not currently possible. However, reasonable results can often be achieved if the topic of conversation is narrowly focused and task oriented. One way of judging our progress to date in giving computers the ability to use language intelligently is to look at the evolution of voice-based search and the rise of “intelligent” assistants. Such a review can also help us understand the limitations of our present tools and techniques.

The first large-scale general purpose (speaker independent, large vocabulary, continuous utterance) deployment of speech recognition technology intended for a wide audience was Google's voice-based web search. Released in 2008 for Android and iPhone smart phones, this technology enabled one to speak a search query, rather than type. While remarkably impressive for its time, Google voice search simply returned a web page of Google search results, exactly as if the speaker had typed the search query.

Two years later, in 2010, Google extended this technology to enable Android phones to perform simple actions based on spoken input. For example, the updated system made it possible to send a text message or email to someone on your contact list by speaking the request (e.g., “Send text message to the prof. Wow! AI has come a long way in a little over fifty years.”). Other actions included calling the phone number of a business by name (e.g., “Call Papa John's in New Orleans”), getting directions (e.g., “Navigate to Louisiana Tech University”), or finding and playing music. This technology required the system to have some limited understanding of what the speaker was asking the phone to do. The implementation was straightforward as the system simply listened for special keywords, such as “navigate”, and then launched the appropriate application passing it any specified parameters, such as “Louisiana Tech University”, when the keyword was detected.

Apple followed suit a year later in the fall of 2011 with the introduction of Siri for the iPhone 4S. Siri, billed as a “personal assistant”, could handle a wider range of tasks than Google Voice Actions and provided for a more natural spoken interface. The interface was far less rigid and didn't require the speaker to remember particular keywords. So, for example, instead needing to say something precise like “Navigate to Louisiana Tech University” one could say “Show me how to get to Louisiana Tech” or “Give me directions to Louisiana Tech.”

Another big innovation was that Siri implemented the ability to respond verbally to some tasks. For example, you could ask Siri “What's my schedule like for tomorrow?” and “she” would access your calendar and go through your list of appointments for the next day. Siri could also schedule an appointment for you and even notify you of conflicts with existing appointments. The system also possessed the ability to verbally respond to a very limited range of factual questions about time and the weather, such as “What time is it in London?” or “Will it rain tomorrow?”

Since weather varies by location, in order to correctly answer such questions Siri needs to know the location to which the user is referring. Unless we explicitly state a location, when we ask such a question we generally mean “right here” which Siri can determine from the iPhone's GPS or cell tower triangulation. However, the implied location can change depending on the context of the conversation.

Siri is smart enough to know that when you ask it “What time is it in London?” followed immediately by “What will the weather be like tomorrow?”, you are probably referring to the weather in London, England. Similarly, if you ask “What will the weather be like in Paris tomorrow?” followed immediately by “in Rome?”, it knows from the context of the conversation that the phrase “in Rome?” means “What will the weather be like in Rome tomorrow?” In order to successfully answer these kinds of questions it is necessary for Siri to possess a rudimentary understanding of conversation and context, something absent from the original version of Google Voice Actions.

Apple further improved Siri with the release of the iPhone 5 in September 2012. The revised version of Siri included expanded knowledge of sports and entertainment, allowing the system to verbally respond to a wider range of questions. For example, in response to “Who is Drew Brees?” Siri replied “Drew Brees currently plays quarterback for the Saints” while showing his picture and player stats. Despite this rather impressive result, the vast majority of questions posed to Siri (as of late 2013) generally return web pages from either a Google search or Wolfram Alpha query.

In July 2012, several months prior to the release of the iPhone 5, Google once again raised the bar with the release of Google Now's integrated voice search which dramatically expanded the scope of questions that can generate a verbal response. By pulling information from Wikipedia and a variety of other sources, reformatting that data, and providing a brief verbal summary, Google Now can provide a verbal response to large numbers of questions over substantial areas of human knowledge. For example, if I ask “When did Man first step foot on the Moon?” Google responds with “July 20, 1969 is one guess based on results below” while displaying the Google search results on which it based its answer.

Over the course of just four years, from 2008 to 2012, voice based interfaces went from quite rare to rather commonplace. As we approach the end of the second decade of the 21st century voice based personal assistants and question answering systems are being used daily by more and more people. However, these systems are still quite limited in their overall capabilities compared to the ultimate goal of constructing computers that can converse intelligently.

IBM's Watson is an example of a question answering system that goes far beyond simple search. It is most famous for winning a two game Jeopardy! tournament in 2011 while competing against Ken Jennings and Brad Rutter, the two most successful humans to ever play the game. During the games, Watson was given exactly the same clues as the human players, with the exception that Watson's clues were in textual form since it could neither see nor hear. After receiving the clue, Watson would decompose it into key words and phrases and then simultaneously run thousands of language analysis algorithms to find statistically related phrases in its repository of stored text (obtained from reference books, web sites, and Wikipedia). Watson then compared these thousands of different results, filtering against clues in the question category (such as the answer is probably a city) to select its final answer, which was generally the most popular result (i.e., the one returned by the greatest number of algorithms). The process, which is far more complex than that used by Google Now and Siri, is often called deep question answering.

Despite recent progress, it is important to understand that neither IBM's Watson nor Google Now's voice search understand the questions they are being asked nor the answers they provide in the way humans do. These systems “simply” compare strings of text in the questions to strings of text in their knowledge bases and then, using statistics along with filters for the type of answer most likely sought (e.g., a name, a date, a place) generate the most likely answer. These systems have no understanding of the meaning of the words in the questions, or the words in the knowledge base, or even the words in the answer that

is generated. Given these constraints, it is truly amazing at the level of performance that these systems have achieved.

Game playing and search techniques

One of the earliest ideas in the quest for machine intelligence (and one that is still used today) is the modeling of decision making as state space search. A state space is a collection of three things:

- (1) A way to represent the state of some system at a particular moment in time by a compact description of the important characteristics of that system;
- (2) A way to determine the legal moves, or transitions, between states of the system; and
- (3) A test to recognize when the goal has been achieved. It is also sometimes necessary to have a separate test to recognize a loss (i.e., a situation from which the goal can never be reached).

Suppose that we wanted to model a board game, such as chess or checkers, using a state space. First, we would need a way to represent the configuration of the game at any point in time. For a board game, this could consist of a simple two-dimensional grid, or array, containing symbols to represent the position of each player's pieces. Next, we would need a description of the rules that apply to moving pieces in the game. For board games, such as chess, these are simply the rules that describe the basic movements of each piece (e.g., the rook may be moved any number of unoccupied squares forward or backward or left or right). While these rules differ from game to game, they are generally quite simple and can be easily mastered. Finally, a test to recognize a winning situation would be necessary. In chess this would amount to a description of checkmate applied to your opponent. Checkmate, applied to your side, also functions as the loss test so that you can know when to quit playing.

Once the game, or problem, has been described in terms of a state space, making a move can be modeled as searching through the space of all possible moves. To be sure, the state space could be huge and intractable to search through! In order to be able to explore deep into a state space without having to deal with an exponentially increasing number of states, heuristic searches are often used. A **heuristic** is a simple measurement that is used as an indicator of some other aspect of the problem that would be difficult or impossible to accurately measure. For example, people often use price as an indicator of quality. Although everyone knows that price and quality are very different things, we sometimes say things like "your new car must have cost you a fortune" when we mean that your new car is very nice. One reason we do things like this is because price is something that is easy to measure, while quality is much more difficult to accurately capture.

A heuristic search is a search that incorporates a heuristic to guide the direction pursued by the search. A simple heuristic for the game of checkers might be:

$$H = \text{NumberOfOpponentPiecesCaptured} - \text{NumberOfYourPiecesLost}$$

This heuristic reflects the fact that, in checkers, the more pieces that you have captured from your opponent and the fewer that you have lost to him or her, generally the better off you are. A heuristic search employing H would first pursue paths that tend to increase H , because these paths are more likely to lead to a win.

A simple heuristic search that can be used in single player games (such as finding your way through a maze) is called "hill climbing." Hill Climbing searches work by:

- (1) Generating all states that are one transition away from the current state;
- (2) Applying a heuristic function to each of these states;
- (3) Moving to the state with the highest heuristic value; and

(4) Repeating this process until the goal is reached.

While hill climbing has some drawbacks, such as the fact that it can get stuck on what are called “local maximums” where every possible next move appears worse than the current state, given a good heuristic function hill climbing can frequently solve problems much more effectively than a blind search.

There are other, more complex, heuristic search algorithms, such as **A*** (pronounced “A star”) and **mini-max**, which overcome many of the shortcomings of hill climbing. Like hill climbing, A* is tailored for single player games. Mini-max is appropriate for two-player games. While additional details about these search procedures are beyond the scope of this lesson, the main thing to remember about heuristic searches is that they generally outperform blind searches because they incorporate knowledge about the problem being solved (or game being played) in order to concentrate effort in directions that are likely to lead to success (rather than searching blindly about for a solution).

Automated reasoning

At about the same time that some computer scientists were studying ways of creating computer systems that could play games (late 1950’s and 1960’s), other computer scientists were thinking about the problem of how to build a system that could “reason.” Much of the early work on automated reasoning focused on constructing computer programs to carry out the rules of symbolic logic. A **symbolic logic**, or formal logic, combines a way of representing information using symbols together with a collection of rules for manipulating those symbols in order to reach logically valid conclusions.

The reason that symbolic logic was of such interest to computer scientists studying artificial intelligence is that these logics are capable of “reasoning” about concepts and ideas based purely on the form of the statements used to represent them. In other words, a symbolic logic can draw logically valid conclusions about a collection of statements based, not on the meaning of those statements, but on the form in which they are written. There are many kinds of symbolic logics, and discussing them is beyond the scope of this lesson.

Neural networks and machine learning

Many of the recent advances in AI, such as progress in speech recognition and automatic translation, have been due to both big data and the resurgence of interest in machine learning techniques (i.e., systems that learn from examples instead of being explicitly programmed for every eventuality).

One particular approach to machine learning, a simple neural network model called the perceptron, is one of the oldest neural network models and the first to gain widespread popularity in the 1960’s. While perceptrons are no longer used in practice today, this relatively simple, biologically inspired computing model provides a good starting point for getting your head around the notion of machine learning systems.

Human brains are composed of trillions of individual nerve cells, called **neurons**. Like all cells, neurons have a nucleus that keeps the neuron alive and functioning. In addition, neurons have a large number of branched protrusions, called **dendrites**, that receive chemical signals from other neurons. Neurons also have long thin fiber-like appendages, called **axons**, down which they send electro-chemical signals. At the end of a neuron’s axon are branch-like structures that come in contact with the dendrites of other neurons. In response to the signals it receives from other neurons, a neuron may fire, sending an electro-chemical pulse down its axon in order to transmit signals to other neurons.

Neurons are not in direct physical contact with one another. Instead, there are tiny gaps, called **synapses**, between the dendrites of one neuron and the axons of others. The signals that pass between neurons must cross these synaptic gaps. This is accomplished by the signaling neuron releasing neurotransmitter chemicals into the synapse. A synapse may be either excitatory or inhibitory. Signals arriving at excitatory synapses increase the likelihood that the receiving neuron will fire. Signals arriving at inhibitory synapses decrease the likelihood of the neuron firing. An interesting feature of biological neurons is that they appear to work in an “all or nothing” fashion (i.e., they either “fire” or they don’t). In other words, neurons do not appear to fire at different strengths; instead, they appear to be either “on” or “off”.

A **perceptron** is a simple processing element that models some of the features of individual neurons. The output of a perceptron is either 0 or 1. The use of a binary output in the perceptron model reflects the fact that biological neurons appear to either “fire” or “not fire”. The inputs to a perceptron, on the other hand, can be arbitrary real numbers: positive, zero, or negative, with fractions allowed.

In order to model the fact that biological neurons possess both excitatory and inhibitory synapses, and the fact that these characteristics appear to vary in strength from synapse to synapse, every input received by the perceptron is multiplied by a weight. As is the case with the inputs themselves, weights are real numbers: they may be positive, zero, or negative, and fractions are allowed.

Once the inputs have been weighted, they are sent to a summation unit that adds them together. The sum of the weighted inputs is then sent to a threshold comparator. Every perceptron will have an internal threshold value that controls how sensitive it is to its inputs. If the weighted sum of a perceptron’s inputs is greater than its internal threshold value, then the perceptron will fire (i.e., generate a 1); otherwise, the perceptron will not fire (i.e., generate a 0).

Because each input is multiplied by a weight, large positive weights increase the effect of a positive input, while smaller positive weights decrease the effect of an input. A weight of zero causes a perceptron to ignore an input. Negative weights, when applied to positive input values, decrease the likelihood of a perceptron firing. In general, when considering positive inputs, positive weights model excitatory synapses (the larger the weight the greater the excitatory effect). Negative weights model inhibitory synapses.

As a perceptron learns, it adjusts the values of its weights and its internal threshold. In fact, the only way a perceptron can “learn” is to adjust its weights and threshold. Those are the only things the perceptron can be said to “know”.

In order for a perceptron to learn, it must be trained. Training involves presenting the perceptron with a group of inputs known as a **training set**. At the beginning of the training process, the perceptron makes random guesses as to whether or not it should fire when presented with a particular input. Whenever the perceptron guesses wrong, its weights will be adjusted by the **perceptron learning rule**. After being adjusted, the training set will be presented to the perceptron again. If the perceptron generates any incorrect results, the learning rule will be applied again. This process will continue until the perceptron correctly classifies all members of the training set.

Before a perceptron is fully trained, it can produce two kinds of incorrect outputs: false positives and false negatives. A false positive occurs when the perceptron fires when it should not have. False negatives occur when the perceptron does not fire when it should have. The perceptron learning rule distinguishes between these two types of errors.

An amazing characteristic of the perceptron learning rule, proven by Rosenblatt in the early 1960's, is that if it is theoretically possible for a perceptron to learn the training set you have presented it with, this algorithm will converge on that solution. In other words, if you run your training examples on the perceptron, modify the weights according to the learning rule, and then repeat the process over and over, eventually all examples will be properly classified. If the training set is **perceptron learnable**, the learning process is guaranteed to eventually halt.

Another remarkable feature of the perceptron learning rule is that it works no matter what values are chosen for the initial weights. Lucky guesses for the initial weights may get you to the solution faster than unlucky guesses, but either way the learning rule will converge on the solution (if there is one) in the end, regardless of how good or bad the initial guesses were.

If we wanted to teach a neural network based system to be able to distinguish between men and women from photographs of their faces, for example, we would digitize a large number of male and female photographs, and then train the network on those photos. What we would expect at the end of the process is that the network has learned to distinguish between male and female facial features. We would be extremely disappointed if the only males and the only females the system could reliably classify were those it had been shown in the training set.

This lesson has covered a lot of material related to AI. It is not meant to be exhaustive nor too technical in nature; rather, it is meant to provide a decent overview of the field of AI and how it has the potential to impact our world.

RPi Activities

Raspberry Pi Activity: Room Adventure...Revolutions

In this activity, you will modify the Room Adventure game that you created and extended in a previous RPi activity. You will need the following items:

- Raspberry Pi B v3 with power adapter;
- LCD touchscreen; and
- Wireless keyboard and mouse with USB dongle.

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 basic premise of the game will not change from the previous activity. Again, the setting of the game is a small “mansion” consisting of four rooms. 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.

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).

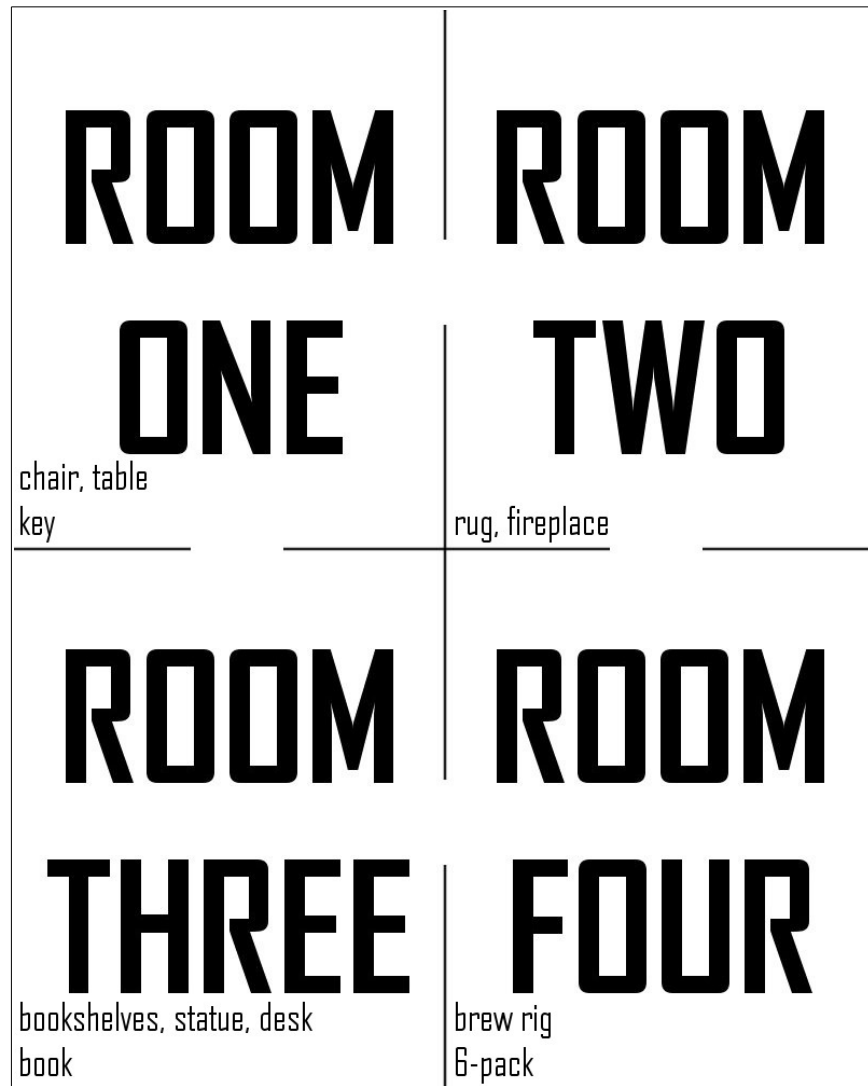
Observable items can provide useful information (once observed) and may reveal new items (some of which can be placed in the player's inventory). In addition, each room may have some items that can be *grabbed* by the player and placed in inventory:

- 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 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.

Here's the layout of the mansion:



The gameplay

The game is text-based, although this activity adds a GUI that integrates the player's input, the status of the room, and an image associated with the room. 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.

As was the case last time, 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

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:

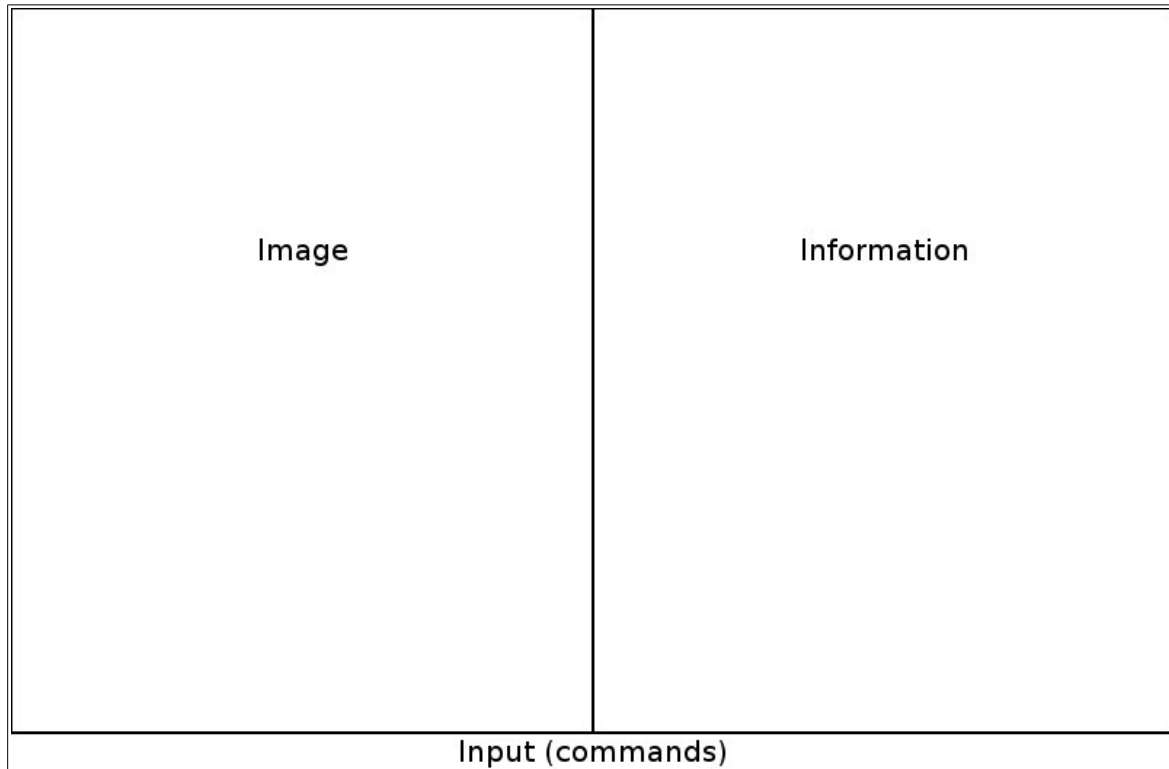
- quit
- exit
- bye
- sionara!

The GUI

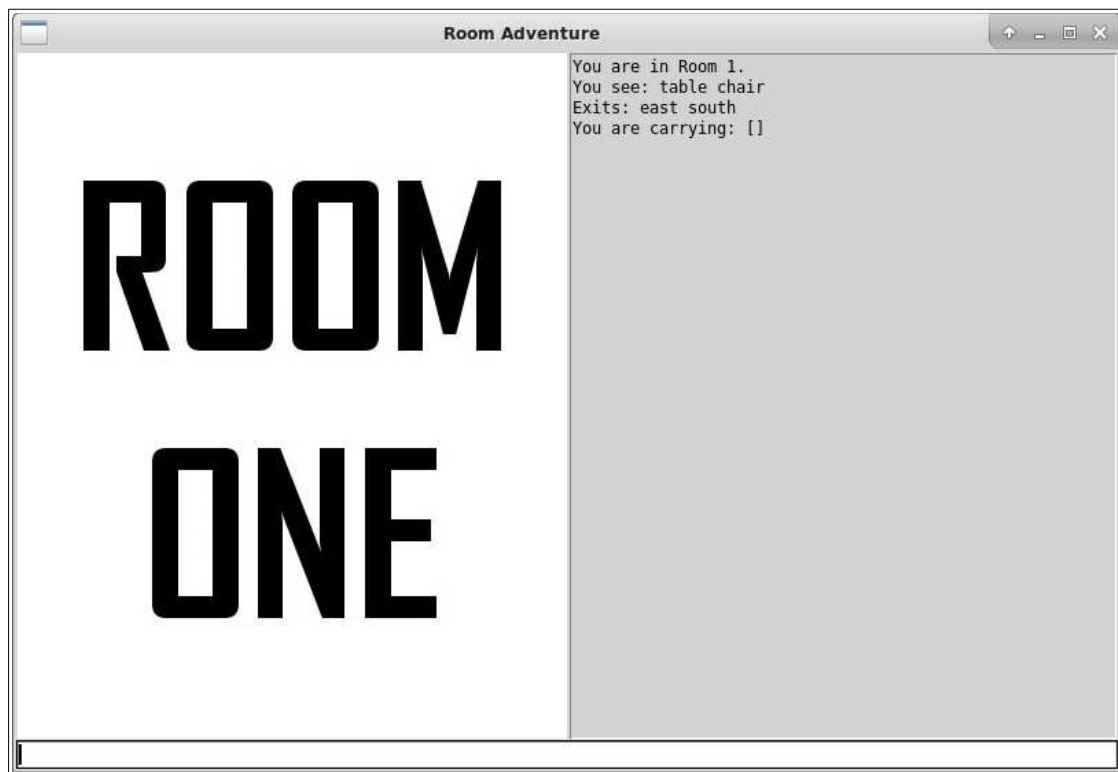
The GUI is split up into three mains parts:

- A text-based representation of the current room that contains its description, items located in the room, and grabbables in the room that can be taken into inventory – along with the player's inventory and some relevant current status;
- An image that provides a static picture of the room; and
- An input action section that takes player input for processing.

Here's a mock-up of the interface:



And here's what the GUI actually looks like once the game is completed and running:



The code

It turns out that a lot of the source code designed in the last Rom Adventure activity can be reused in this activity as well. Some of it, however, will need to be moved around and/or redesigned so that the GUI can work seamlessly with the backend (e.g., rooms, items, inventory, player input, etc).

Instead of starting with the source code from the previous activity, we will simply build the new Room Adventure game from scratch by integrating what we can from the old version. So let's begin with the main part of the program (which turns out to be just a few lines of code!):

```
#####  
# Name:  
# Date:  
# Description:  
#####  
from Tkinter import *  
  
...the rest of the code will go here...  
  
#####  
# the default size of the GUI is 800x600  
WIDTH = 800  
HEIGHT = 600  
  
# create the window  
window = Tk()  
window.title("Room Adventure")  
  
# create the GUI as a Tkinter canvas inside the window  
g = Game(window)  
# play the game  
g.play()  
  
# wait for the window to close  
window.mainloop()
```

Note that, in the final program, this source code will be located at the bottom. To start things off, we import Tkinter so that the GUI can be created. In fact, the GUI is created much like the examples in the lesson on GUIs. The width and height of the GUI are defined as constants. A new instance of the class **Tk** is then created. This *window* is given an appropriate title. The rest of the GUI components are defined in a **Game** class, which is instantiated as the variable *g*. The **Game** class will have a *play* method that will kick things off and officially start the game. Of course, the window must remain on the player's desktop until it is closed.

The remainder of the code will be inserted in between the header (and import statements) and the main part of the program. Let's begin with the **Room** class (which happens to be almost exactly the same as the modified version that uses dictionaries instead of parallel lists). Changes are highlighted:

```
# the room class  
class Room(object):
```

```

# the constructor
def __init__(self, name, image):
    # rooms have a name, an image (the name of a file),
    # 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.image = image
    self.exits = {}
    self.items = {}
    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 image(self):
    return self._image

@image.setter
def image(self, value):
    self._image = value

@property
def exits(self):
    return self._exits

@exits.setter
def exits(self, value):
    self._exits = value

@property
def items(self):
    return self._items

@items.setter
def items(self, value):
    self._items = 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
    # dictionary
    self._exits[exit] = 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 description to the appropriate
    # dictionary
    self._items[item] = 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.keys():
        s += item + " "
    s += "\n"

    # next, the exits from the room
    s += "Exits: "

```

```

        for exit in self.exits.keys():
            s += exit + " "

    return s

```

The only change to the **Room** class is the addition of a new instance variable that represents a room's image. As such, the constructor has minor changes, and an accessor and mutator is also provided for the new instance variable.

The Game class is quite involved, so we'll build it a little at a time. Let's begin with a shell:

```

# the game class
# inherits from the Frame class of Tkinter
class Game(Frame):
    # the constructor
    def __init__(self, parent):
        # call the constructor in the superclass
        Frame.__init__(self, parent)

    # creates the rooms
    def createRooms(self):
        ...

    # sets up the GUI
    def setupGUI(self):
        ...

    # set the current room image
    def setRoomImage(self):
        ...

    # sets the status displayed on the right of the GUI
    def setStatus(self, status):
        ...

    # play the game
    def play(self):
        # add the rooms to the game
        self.createRooms()
        # configure the GUI
        self.setupGUI()
        # set the current room
        self.setRoomImage()
        # set the current status
        self.setStatus("")

    # processes the player's input
    def process(self, event):
        ...

```

Overall, the **Game** class is almost self-explanatory. It inherits from Tkinter's **Frame** class in order to support GUI components. By default, the class variables `WIDTH` and `HEIGHT` define a GUI that is 800 pixels wide by 600 pixels high. The constructor simply calls the constructor of its superclass (Tkinter's **Frame** class). Several method *stubs* (i.e., temporary templates) are provided to create the rooms (these are the actual room objects that provide backend support for the game), to setup the GUI components, to set the room image on the GUI (as the player moves around the mansion), to set the current status (as the player's input is processed), and to process the player's input. These functions will be covered later.

The play function is brief. It first creates the rooms, sets up the GUI, sets the current room image, and sets the current status. This is all that's required to begin playing the game. Creating the rooms will setup each room's exits, images, items, and grabbables. Setting up the GUI will create the GUI components and display the GUI. Setting the room image will display the appropriate image to the left of the GUI. Setting the current status will display the appropriate status to the right of the GUI.

Let's continue by implementing the `createRooms` function:

```
# creates the rooms
def createRooms(self):
    # 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)

    # create the rooms and give them meaningful names and an
    # image in the current directory
    r1 = Room("Room 1", "room1.gif")
    r2 = Room("Room 2", "room2.gif")
    r3 = Room("Room 3", "room3.gif")
    r4 = Room("Room 4", "room4.gif")

    # 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
        Game.currentRoom = r1

        # initialize the player's inventory
        Game.inventory = []

```

This updated function is almost exactly the same as in the last Room Adventure program. Differences have been highlighted. Since the function is in the **Game** class, then the variable `self` must be passed in to it. This is standard procedure for most methods in classes.

When the rooms are instantiated, a string representing the file name of an associated image must also be passed in. This is due to the change in the constructor of the **Room** class (which now takes this as a parameter so that an image can be associated with a room). At the end of the function, the current room is set (as `r1`). The `currentRoom` variable is now a class variable (i.e., valid for the entire class). As such, it must be referred to using the class name: `Game.currentRoom`. Finally, the player's inventory is initialized here (also a class variable).

The next function that we'll implement is `setupGUI`:

```

        # sets up the GUI
        def setupGUI(self):
            # organize the GUI
            self.pack(fill=BOTH, expand=1)

            # setup the player input at the bottom of the GUI

```

```

# the widget is a Tkinter Entry
# set its background to white and bind the return key to the
# function process in the class
# push it to the bottom of the GUI and let it fill
# horizontally
# give it focus so the player doesn't have to click on it
Game.player_input = Entry(self, bg="white")
Game.player_input.bind("<Return>", self.process)
Game.player_input.pack(side=BOTTOM, fill=X)
Game.player_input.focus()

# setup the image to the left of the GUI
# the widget is a Tkinter Label
# don't let the image control the widget's size
img = None
Game.image = Label(self, width=WIDTH / 2, image=img)
Game.image.image = img
Game.image.pack(side=LEFT, fill=Y)
Game.image.pack_propagate(False)

# setup the text to the right of the GUI
# first, the frame in which the text will be placed
text_frame = Frame(self, width=WIDTH / 2)
# the widget is a Tkinter Text
# disable it by default
# don't let the widget control the frame's size
Game.text = Text(text_frame, bg="lightgrey", state=DISABLED)
Game.text.pack(fill=Y, expand=1)
text_frame.pack(side=RIGHT, fill=Y)
text_frame.pack_propagate(False)

```

The comments in the source code explain what's going on. Basically, a Tkinter **Frame** will contain the three main GUI components. We make sure that the frame is the same size as the window. Next, the player input section at the bottom of the GUI is created. The Tkinter widget that is used is the **Entry** widget. It is configured and bound to the `process` function (that we'll cover later). That is, the desired behavior is to invoke the `process` function when the player types an action and presses Enter/Return. In addition, this widget is given focus so that the player doesn't have to keep clicking on it to type an action.

The left of the GUI is then setup. In Tkinter, we can display images in a **Label** widget. We set its width to half of the width of the window and add its image (initially `None`). The statement `Game.image.pack_propagate(False)` ensures that no component or widget inside the label will affect its size. That is, the size of the left side of the GUI will remain fixed.

Finally, the right of the GUI is setup. To properly display the current game status, we place a Tkinter **Text** widget inside of a Tkinter **Frame**. We must also ensure that the size of this widget remains fixed.

Next, let's work on the `setRoomImage` function that sets the appropriate image on the left of the GUI:

```
# set the current room image
def setRoomImage(self):
    if (Game.currentRoom == None):
        # if dead, set the skull image
        Game.img = PhotoImage(file="skull.gif")
    else:
        # otherwise grab the image for the current room
        Game.img = PhotoImage(file=Game.currentRoom.image)

    # display the image on the left of the GUI
    Game.image.config(image=Game.img)
    Game.image.image = Game.img
```

This function first checks if the current room is valid. If set to `None`, the implication is that the player has died (i.e., jumped out of the window by going south in room 4). Either way, the appropriate image is displayed on the **Label** widget on the left side of the GUI

Now, let's work on the `setStatus` function that displays the current game status on the right side of the GUI:

```
# sets the status displayed on the right of the GUI
def setStatus(self, status):
    # enable the text widget, clear it, set it, and disabled it
    Game.text.config(state=NORMAL)
    Game.text.delete("1.0", END)
    if (Game.currentRoom == None):
        # if dead, let the player know
        Game.text.insert(END, "You are dead. The only thing
you can do now is quit.\n")
    else:
        # otherwise, display the appropriate status
        Game.text.insert(END, str(Game.currentRoom) +\
            "\nYou are carrying: " + str(Game.inventory) +\
            "\n\n" + status)
    Game.text.config(state=DISABLED)
```

Since the **Text** widget is disabled so that the player cannot edit it, it must first be enabled in so that its contents can be changed. Its current text is then deleted, the new status is added, and the widget is once again disabled. Again, if the player is dead, the current room is set to `None`, and we can easily determine what the current status should be.

The last function that needs to be implemented is the `process` function. It serves as the driver that continuously processes the player's input. In fact, it is invoked each time the player presses Enter/Return in the player action input section at the bottom of the GUI:

```
# processes the player's input
def process(self, event):
    # grab the player's input from the input at the bottom of
    # the GUI
    action = Game.player_input.get()
    # set the user's input to lowercase to make it easier to
    # compare the verb and noun to known values
    action = action.lower()
    # set a default response
    response = "I don't understand. Try verb noun. Valid verbs
are go, look, and take"

    # exit the game if the player wants to leave (supports quit,
    # exit, and bye)
    if (action == "quit" or action == "exit" or action == "bye" \
        or action == "sionara!"):
        exit(0)

    # if the player is dead if goes/went south from room 4
    if (Game.currentRoom == None):
        # clear the player's input
        Game.player_input.delete(0, END)
        return

    # 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]

        # the verb is: go
        if (verb == "go"):
            # set a default response
            response = "Invalid exit."

            # check for valid exits in the current room
            if (noun in Game.currentRoom.exits):
                # if one is found, change the current room to
                # the one that is associated with the
                # specified exit
                Game.currentRoom = \
```

```

        Game.currentRoom.exits[noun]
        # set the response (success)
        response = "Room changed."
# 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
    if (noun in Game.currentRoom.items):
        # if one is found, set the response to the
        # item's description
        response = Game.currentRoom.items[noun]
# 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 Game.currentRoom.grabbables:
        # a valid grabbable item is found
        if (noun == grabbable):
            # add the grabbable item to the player's
            # inventory
            Game.inventory.append(grabbable)
            # remove the grabbable item from the
            # room
            Game.currentRoom.delGrabbable(grabbable)
            # set the response (success)
            response = "Item grabbed."
            # no need to check any more grabbable
            # items
            break

# display the response on the right of the GUI
# display the room's image on the left of the GUI
# clear the player's input
self.setStatus(response)
self.setRoomImage()
Game.player_input.delete(0, END)

```

You may notice that this function is quite similar to the source code in the last version of the Room Adventure game (that uses dictionaries). Changes are **not** highlighted because there are simply too many of them. In addition, much of the code has been slightly modified and reordered.

The function first grabs the player's input from the bottom of the GUI and converts it to lowercase. A default response is set. If the player wishes to quit, we do so. If the player is dead, we clear the player's

input and do nothing. At this point, the image on the left of the GUI is the skull, and the only thing that the player can do is to quit the game.

Next, the player's input is split into two words: verb noun. The verb is then tested. If the player wishes to go in the direction of another room, it becomes the current room; otherwise, an error response is generated. If the player wishes to look at a valid item in the room, the item's description is set as the response. If the player wishes to take a valid item in the room, the item is added to the player's inventory. All of these things occurred in the previous version of the Room Adventure game.

Finally, the response is set as the current status and is updated in the **Text** widget, the room's image is updated in the **Label** widget, and the player's input is cleared in the **Entry** widget.

Homework: Room Adventure...Revolutions

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 re-implement the Room Adventure improvements that you made in your text-based version of the game – in this new GUI version of the game. Of course, feel free to add new improvements! Note that this component is worth **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).

Clearly, you will need to add images for each room. Note that images should be 400x550 pixels. The height of 550 pixels is slightly flexible (e.g., 500 or 600 pixels should also work). **Optionally**, you can add sound to the game. An idea would be to have a different sound play in each room! Note that this is a bit more difficult and is entirely optional.

Make sure to put an appropriate header at the top of your program and to appropriately comment your source code as necessary. Since your game will now include images for each room, submit your source code, images, and any other required file(s) (e.g., sound files) **as a single ZIP archive**. If you include sound files, please make sure that they are compressed and/or optimized. **There is a 10MB limit on the size of the ZIP archive. Submissions that are greater than this size will not be accepted!**

Raspberry Pi Activity: Paper Piano

In this activity, you will implement a simple piano using nothing but pieces of paper and pencil lead as the keys! Well, almost. Instead of using push-button switches (as in previous activities), you will use capacitive touch switches. These switches work by using your body's capacitance. **Capacitance** is the ability of something to store an electrical charge (which your body can do!). When you touch the switch, the capacitance is increased – and the switch is triggered. The electrical signal will then be amplified by a transistor so that the RPi can detect it as a switch press on an input pin. Transistors will be discussed later in this activity.

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;
- LEDs, resistors, switches, and jumper wires provided in your kit;
- Transistors;
- Paper and pencil (#2 works well) for the piano keys; and
- Scotch tape to tape jumper wires to the paper piano keys (provided to you).

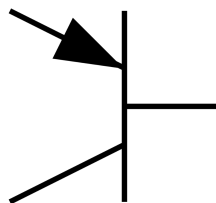
Regarding the electronic components, you will need the following:

- 1x RGB LED;
- 2x push-button switches;
- 4x S8550 PNP transistors;
- 24x jumper wires.

The activity is broken down into three parts. In the first part, you will just implement a single piano key that will play a middle C note. In the second part, you will extend the piano to include four keys so that four notes can be played (C, E, G, and B). In the final part, you will add the functionality to record notes played – and play them back!

Part one: a single piano key

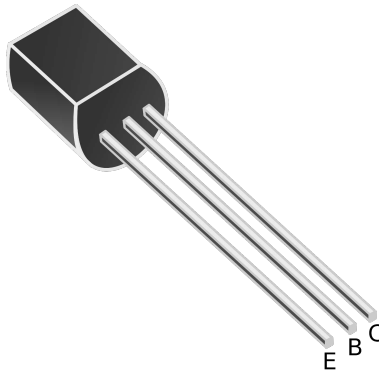
For this part of the activity, you will implement a circuit that includes a new component that has not yet been used in an activity. In a circuit diagram, its symbol looks like this:



This is the symbol for a PNP transistor. A **transistor** is an electronic device that is used to amplify and/or switch electrical signals. It has three terminals: a base, an emitter, and a collector. Voltage or current applied to one pair of terminals changes the current through the other pair. Today's electronic devices are fundamentally based on transistors.

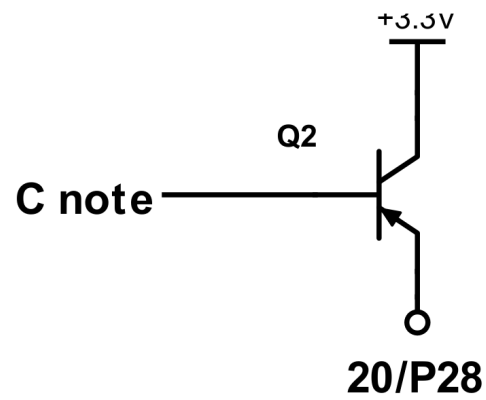
A thorough discussion of transistors is beyond the scope of this activity. However, the way that the transistors are used in this activity is simple enough. The base will be connected to a piece of paper with some graphite on it (the middle C note), the collector will be connected to +3.3V, and the emitter will be connected to an input pin on the RPi. When the base is not touched, the circuit is open. That means that electric current cannot pass from the collector to the emitter. In order to turn on the transistor, a little bit of power must be supplied to the base. Our body contains a little bit of electric current (via capacitance) which is enough to turn the transistor on. When the note (and by consequence the base) is touched, this little amount of current will be greatly amplified and turn on the transistor, making it act like a switch.

The transistor that you will use in this activity is an S8550 in what is called a TO-92 package. It looks like this (note that the flat side is to the front, and this orients the legs so that the emitter, base, and collector are easy to identify):

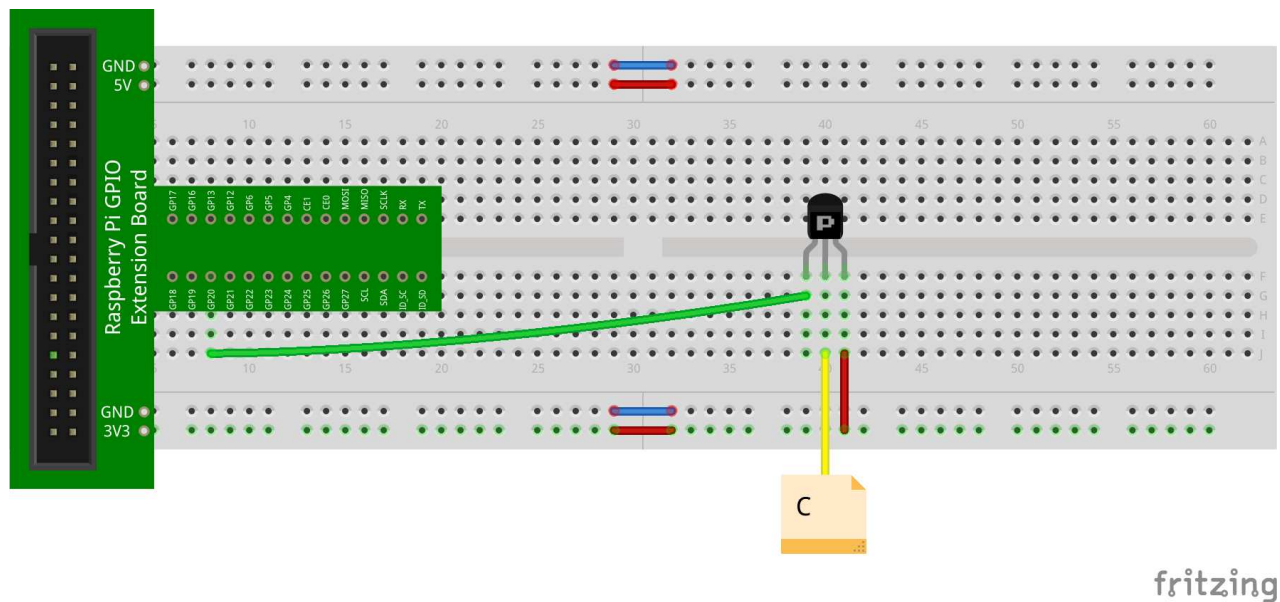


Note the labels for the transistor's legs in the image above. When the flat side of the transistor is facing the front, the emitter (labeled E) is the left leg, the base (labeled B) is the center leg, and the collector (labeled C) is the right leg. When inserting the transistor into the center of the breadboard, make sure that each leg is in a separate column.

Implement the following circuit:

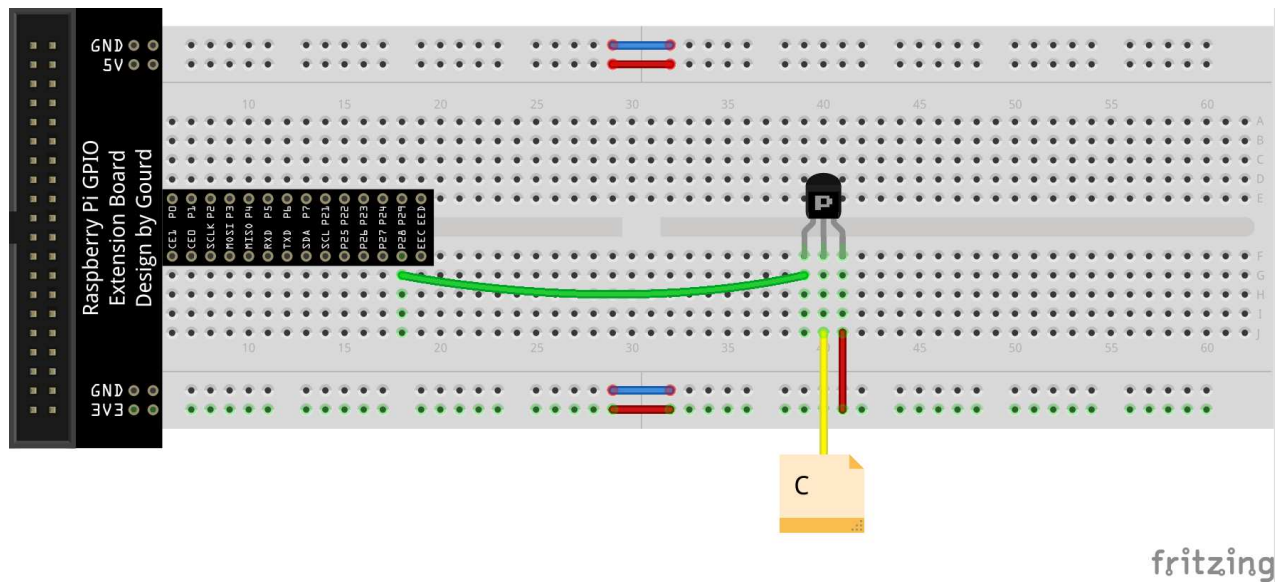


Here's one way to layout this circuit:



fritzing

If you have the black GPIO interface board, layout the circuit as follows instead:

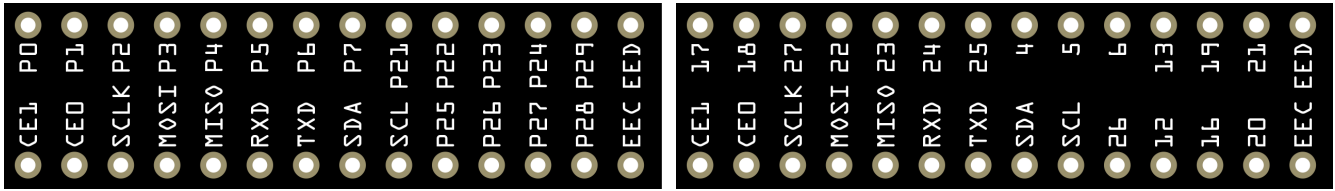


fritzing

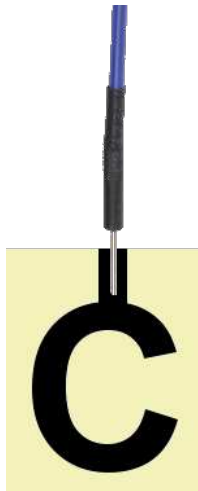
Recall that there are actually **three** different pin numbering schemes in use with GPIO pins on the RPi: (1) the **physical** pin order on the RPi; (2) the numbering assigned by the manufacturer of the **Broadcom** chip on the RPi; and (3) an older numbering assigned by an early RPi user who developed 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
		GND	9	10	RXD	16
17	0	GPIO.0	11	12	GPIO.1	1
27	2	GPIO.2	13	14	GND	
22	3	GPIO.3	15	16	GPIO.4	4
		3V3	17	18	GPIO.5	5
10	12	MOSI	19	20	GND	
9	13	MISO	21	22	GPIO.6	6
11	14	SCLK	23	24	CE0	10
		GND	25	26	CE1	11
0	30	SDA.0	27	28	SCL.0	31
5	21	GPIO.21	29	30	GND	
6	22	GPIO.22	31	32	GPIO.26	26
13	23	GPIO.23	33	34	GND	
19	24	GPIO.24	35	36	GPIO.27	27
26	25	GPIO.25	37	38	GPIO.28	28
		GND	39	40	GPIO.29	29

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 emitter of the transistor is connected to **GP20** (which refers to BCM pin **20** on the RPi and **P28** on the black GPIO interface). The collector is connected to +3.3V, and the base is connected to a piece of paper with some graphite on it. Here's an example of how a jumper wire can be connected (with some Scotch tape) to the graphite:



Make sure that the collector is connected to 3.3V (and not 5V)!

Next, let's take a look at some Python source code that will implement the simple paper piano that plays a single middle C note (for now). The first step is to import the required libraries:

```
import RPi.GPIO as GPIO
from time import sleep
import pygame
from array import array
```

The GPIO library will be used to support GPIO on the RPi. The sleep function will be used to temporarily “sleep” while waiting for piano keys to be pressed and released. The pygame library will be used to generate and play the notes. Finally, the array library will be used to create arrays of note samples. In Python, arrays are just like lists, except that the type of data stored in them is constrained. In the case of this activity, we will store a note's samples in 16-bit signed numbers. A thorough discussion of how the notes are actually generated is beyond the scope of this activity. A few details will be provided later in the activity; however, you are encouraged to research on your own the technical aspects of how the pygame library can generate notes.

Next, we specify several constants and a Note class (which inherits from pygame's mixer's Sound class):

```
MIXER_FREQ = 44100
```

```

MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
    # note that volume ranges from 0.0 to 1.0
    def __init__(self, frequency, volume):
        self.frequency = frequency
        # initialize the note using an array of samples
        pygame.mixer.Sound.__init__(self, \
            buffer=self.build_samples())
        self.set_volume(volume)

    # builds an array of samples for the current note
    def build_samples(self):
        # calculate the period and amplitude of the note's wave
        period = int(round(MIXER_FREQ / self.frequency))
        amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
        # initialize the note's samples (using an array of
        # signed 16-bit "shorts")
        samples = array("h", [0] * period)

        # generate the note's samples
        for t in range(period):
            if (t < period / 2):
                samples[t] = amplitude
            else:
                samples[t] = -amplitude

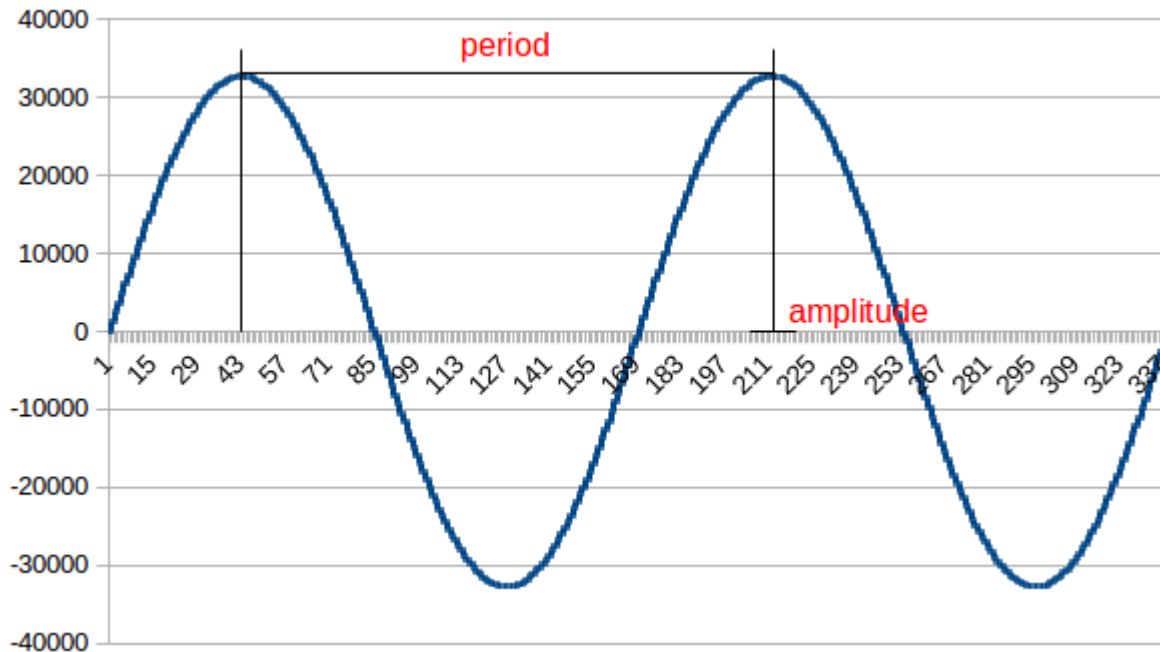
        return samples

```

The constants setup the mixer to sample 44,100 times per second. The minimum and maximum values for each sample is represented by a 16-bit signed number (i.e., -32,768 to 32,767). The mixer will have a single channel (mono) and a buffer of 1KB.

This forms the core of generating the notes to play. In short, a note is initialized with a frequency and volume. The frequency of a middle C, for example, is 261.6 Hz. Frequency is the number of cycles per second and controls a note's pitch. To make sure that we hear the notes well, we will blast them at full volume (1.0). Start with your speakers set to low and increase the volume as needed. If the speakers are too loud, the sound will sometimes be distorted and appear to skip.

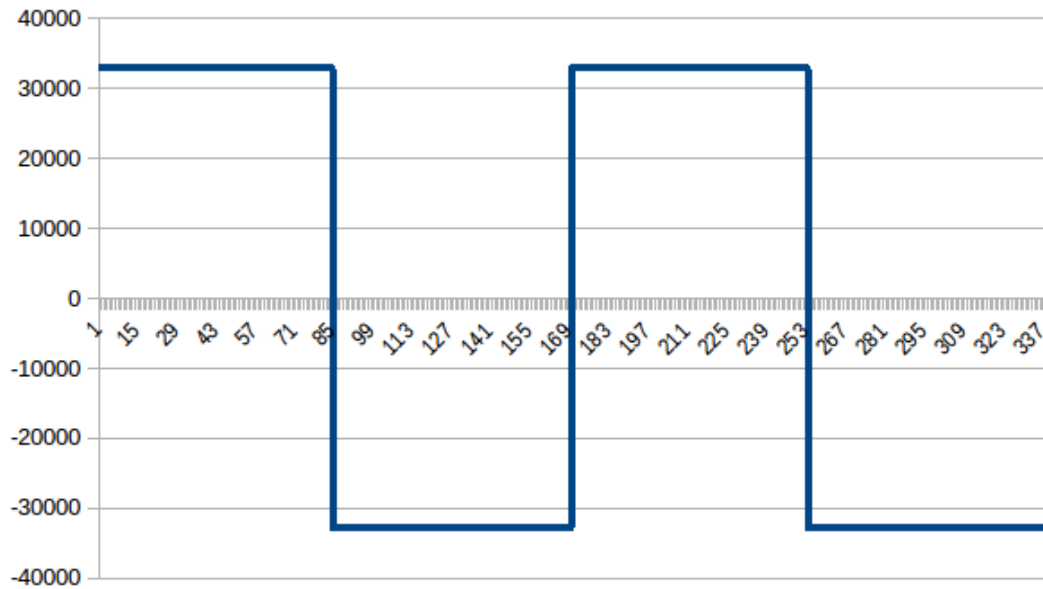
Each note will need samples for the mixer to generate and play them. The samples are generated from the mixer's sampling frequency (set at 44.1 KHz). The sampling frequency (or sample rate) is the number of samples per second in some sound. 44.1 KHz is considered CD quality. Notes are just waves with a period (the length from one peak to the next), and an amplitude (the height from the center line to the peak):



Formally, a period is just the number of seconds per cycle. In our program, we will tweak this meaning slightly to the number of samples within the sampling rate (of 44.1 KHz). For the middle C (at a frequency of 261.6 Hz), for example, the number of samples that make up a single period is 169 (well, 168.58 – but we'll round up). The number of samples in a period is calculated as the ratio of the sampling frequency to the note's frequency. We want to generate a number of samples equal to the period of the note's wave. The mixer will then play this over and over again, for the duration of the note. The amplitude of a wave affects how loud the note is.

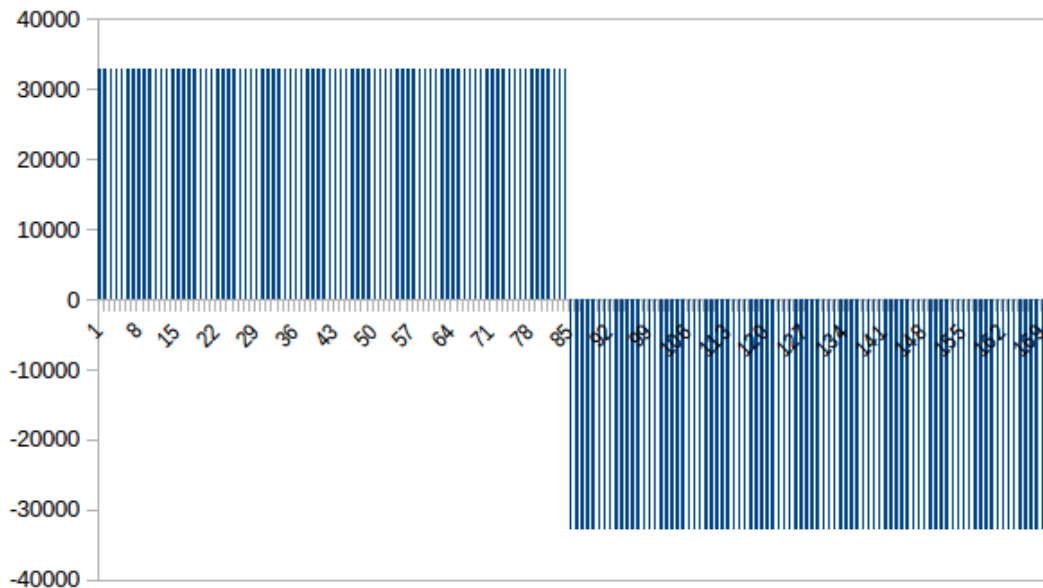
It is not possible to represent a continuous sinusoidal wave in a computing machine (since it is discrete!). Therefore, we have to approximate without creating too much of a loss. One modification allows the note to keep most of its characteristics, albeit perhaps a bit more harsh and bright. This square version of the wave (called a square wave) results in a maximum amplitude for half of the period, and the negative of the maximum amplitude for the other half of the period.

The following shows the square wave approximation of the sinusoidal wave shown above:



Note how much sharper it is. In the end, however, the result is a sound that is near the intended one. In terms of pitch (which is affected by the wave's frequency), it is exact. The sound is actually a bit louder, since more points along the wave are at the frequency limit ($-32,768$ to $32,767$). This limit is due to the size of the mixer (set at 16-bits signed).

For this activity, we will create a square wave and fill the samples with half of the maximum amplitude and half of the negative of the maximum amplitude as follows:



The array of samples is initialized in the following statement:

```
samples = array("h", [0] * period)
```

In Python, arrays are just objects that are defined in a class. The first parameter specified when instantiating an array is the type of data that will be stored in it. There are many different types, and `h` restricts the array to 16-bit signed numbers. The second parameter is just a list of zeros, the number of which is determined by the period of the note. A middle C with a frequency of 261.6 Hz will have a period of 169 samples with a sample rate of 44.1 KHz. That is, a middle C takes up 169 samples per period in the 44.1 K samples per second.

Next, we'll define two functions that do the work of waiting for notes to be pressed and released. In addition, we initialize the pygame library, setup GPIO, and create the middle C note:

```
# waits until a note is pressed
def wait_for_note_start():
    while (not GPIO.input(key)):
        sleep(0.01)

# waits until a note is released
def wait_for_note_stop():
    while (GPIO.input(key)):
        sleep(0.1)

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
    MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pin and frequency for a C note
key = 20
freq = 261.6

# setup the input pin
GPIO.setup(key, GPIO.IN, GPIO.PUD_DOWN)

# create the actual C note
note = Note(freq, 1)
```

The `wait_for_note_start` function sleeps until the note is pressed. The `wait_for_note_stop` function does the opposite: it sleeps until the note is released. As mentioned earlier, P28=BCM 20 is used to detect the note press and release. The frequency for a middle C (261.6 Hz) is specified, and the note itself is generated as an instance of the `Note` class.

Lastly, the main part of the program:

```
# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
    while (True):
        # play a note when pressed...until released
        wait_for_note_start()
        note.play(-1)
        wait_for_note_stop()
        note.stop()
except KeyboardInterrupt:
    # reset the GPIO pins
    GPIO.cleanup()
```

It's fairly simple, actually. Also note that it is just a driver. Until the user presses Ctrl+C, the program waits for a note to be pressed, plays the note, waits for the note to be released, and stops playing the note. The -1 argument passed to the note's play function means that it will play indefinitely (well, at least until its stop function is called).

Implement the circuit and program above, then test it! For completeness, here is the program in its entirety:

```
#####
# Name:
# Date:
# Description: Paper piano (v1).
#####
import RPi.GPIO as GPIO
from time import sleep
import pygame
from array import array

MIXER_FREQ = 44100
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
    # note that volume ranges from 0.0 to 1.0
    def __init__(self, frequency, volume):
        self.frequency = frequency
        # initialize the note using an array of samples
        pygame.mixer.Sound.__init__(self,\
```



```

        buffer=self.build_samples())
    self.set_volume(volume)

# builds an array of samples for the current note
def build_samples(self):
    # calculate the period and amplitude of the note's wave
    period = int(round(MIXER_FREQ / self.frequency))
    amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
    # initialize the note's samples (using an array of
    # signed 16-bit "shorts")
    samples = array("h", [0] * period)

    # generate the note's samples
    for t in range(period):
        if (t < period / 2):
            samples[t] = amplitude
        else:
            samples[t] = -amplitude

    return samples

# waits until a note is pressed
def wait_for_note_start():
    while (not GPIO.input(key)):
        sleep(0.01)

# waits until a note is released
def wait_for_note_stop():
    while (GPIO.input(key)):
        sleep(0.1)

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
    MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pin and frequency for a C note
key = 20
freq = 261.6

# setup the input pin
GPIO.setup(key, GPIO.IN, GPIO.PUD_DOWN)

```

```

# create the actual C note
note = Note(freq, 1)

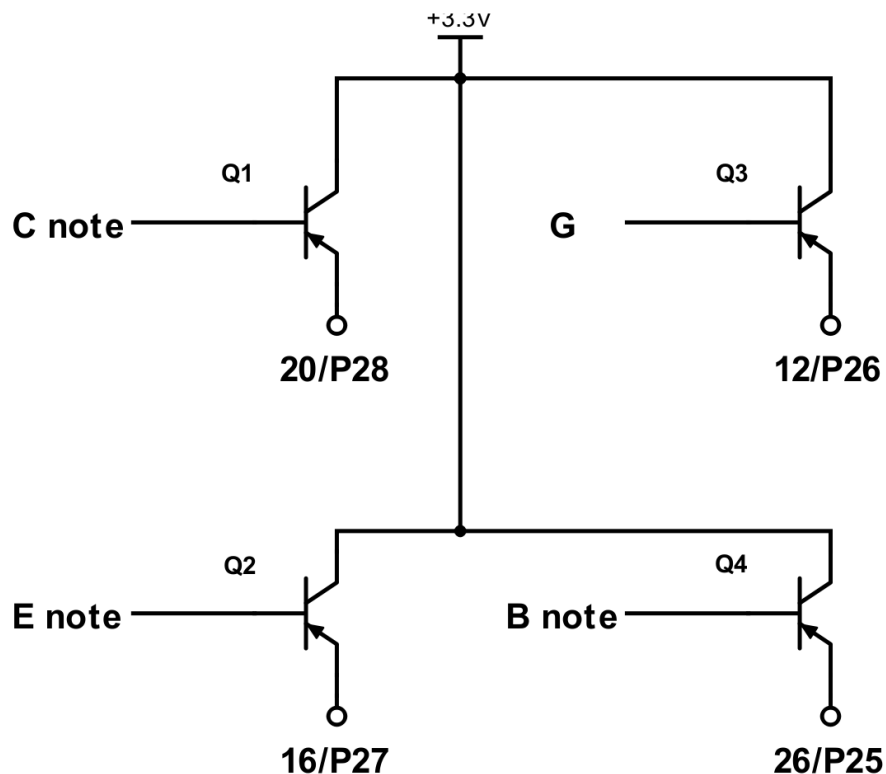
# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
    while (True):
        # play a note when pressed...until released
        wait_for_note_start()
        note.play(-1)
        wait_for_note_stop()
        note.stop()
except KeyboardInterrupt:
    # reset the GPIO pins
    GPIO.cleanup()

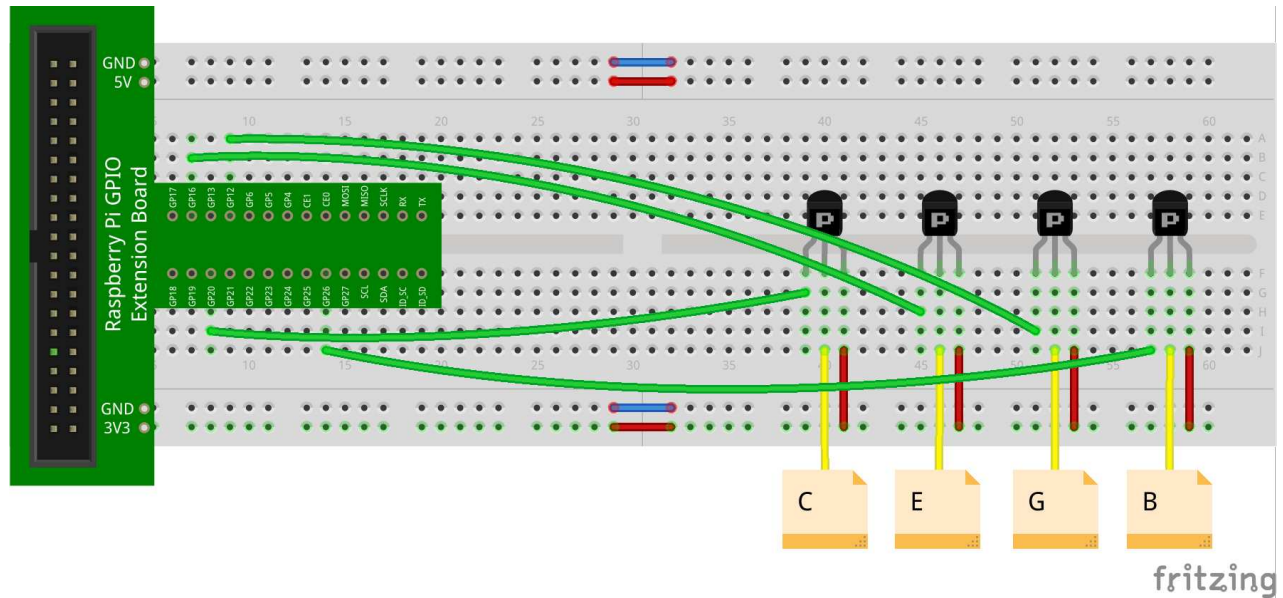
```

Part two: adding more piano keys

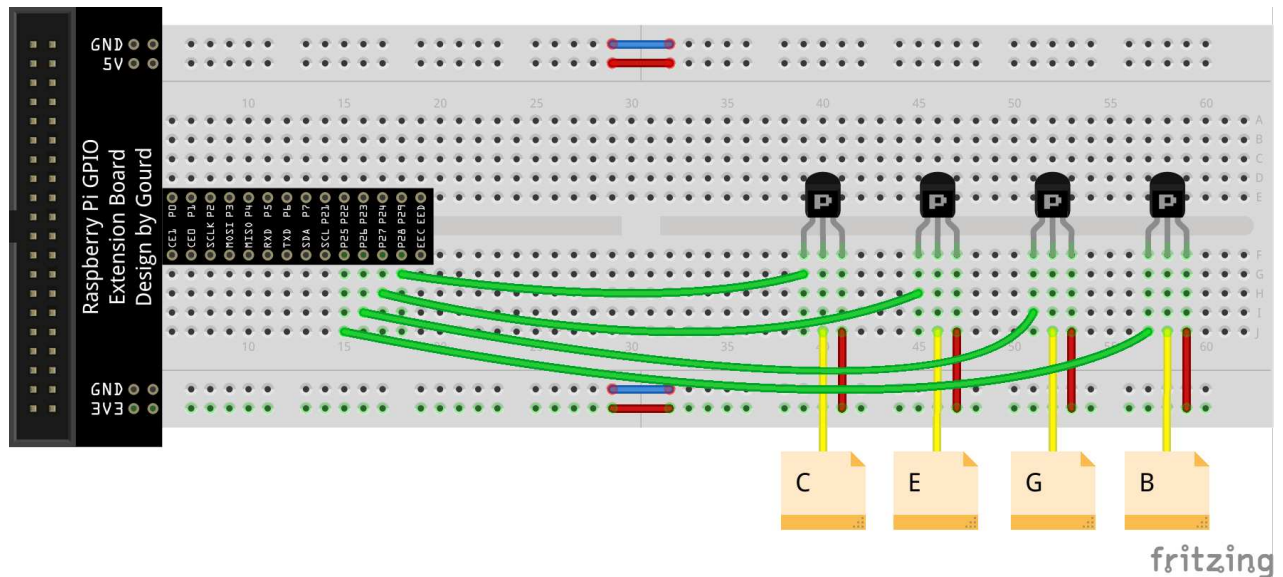
In this part of the activity, we will add three more piano keys (E, G, and B). The process is exactly the same; however, we will slightly modify the code to handle the extra notes. Here's the circuit:



Here's one way to layout this circuit:



If you have the black GPIO interface board, layout the circuit as follows instead:



Note that only three transistors have been added (and their respective connections to GPIO pins, +3.3V, and jumper wires connected to pieces of paper with some graphite on them). **It is recommended to use the longest jumper wires to connect the base of the transistors to the pieces of paper.** This makes playing the piano a bit easier. The emitter of the transistor used for the E note is connected to GP16 (P27). Similarly, GP12 (P26) is used for the G note, and GP26 (P25) is used for the B note.

Again, make sure that the collector of each transistor is connected to 3.3V (and not 5V)!

Once you have implemented the additions to the circuit, you will need to change the code to handle the extra notes. First, the extra GPIO pins and the note frequencies are added:

Replace:

```
# setup the pin and frequency for a C note
key = 20
freq = 261.6
```

With:

```
# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []
```

Note that we are using lists to represent the GPIO pins, note frequencies, and the generated notes themselves.

Next, we change the setup of the GPIO pins:

Replace:

```
# setup the input pin
GPIO.setup(key, GPIO.IN, GPIO.PUD_DOWN)
```

With:

```
# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)
```

Then, we create the new notes as instances of the Note class:

Replace:

```
# create the actual C note
note = Note(freq, 1)
```

With:

```
# create the actual notes
for n in range(len(freqs)):
    notes.append(Note(freqs[n], 1))
```

Next, we slightly modify the main part of the program so that the note that is pressed is saved, and it's release can be properly detected:

Replace:

```
# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
    while (True):
        # play a note when pressed...until released
        wait_for_note_start()
        note.play(-1)
        wait_for_note_stop()
        note.stop()
```

```
except KeyboardInterrupt:
```

With:

```
# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
```

```
try:
```

```
    while (True):
```

```
        # play a note when pressed...until released
```

```
        key = wait_for_note_start()
```

```
        notes[key].play(-1)
```

```
        wait_for_note_stop(keys[key])
```

```
        notes[key].stop()
```

```
except KeyboardInterrupt:
```

Finally, we must also modify the `wait_for_note_start` and `wait_for_note_stop` functions as follows, so that every note press and release can be detected:

Replace:

```
# waits until a note is pressed
```

```
def wait_for_note_start():
```

```
    while (not GPIO.input(key)):
```

```
        sleep(0.01)
```

```
# waits until a note is released
```

```
def wait_for_note_stop():
```

```
    while (GPIO.input(key)):
```

```
        sleep(0.1)
```

With:

```
# waits until a note is pressed
```

```
def wait_for_note_start():
```

```
    while (True):
```

```
        for key in range(len(keys)):
```

```
            if (GPIO.input(keys[key])):
```

```
                return key
```

```
        sleep(0.01)
```

```
# waits until a note is released
```

```
def wait_for_note_stop(key):
```

```
    while (GPIO.input(key)):
```

```
        sleep(0.1)
```

Implement the changes specified above and make sure that your paper piano can play all four notes properly.

For reference, here is the entire modified source code:

```
#####
# Name:
```

```

# Date:
# Description: Paper piano (v2).
#####
import RPi.GPIO as GPIO
from time import sleep
import pygame
from array import array

MIXER_FREQ = 44100
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
    # note that volume ranges from 0.0 to 1.0
    def __init__(self, frequency, volume):
        self.frequency = frequency
        # initialize the note using an array of samples
        pygame.mixer.Sound.__init__(self, \
            buffer=self.build_samples())
        self.set_volume(volume)

    # builds an array of samples for the current note
    def build_samples(self):
        # calculate the period and amplitude of the note's wave
        period = int(round(MIXER_FREQ / self.frequency))
        amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
        # initialize the note's samples (using an array of
        # signed 16-bit "shorts")
        samples = array("h", [0] * period)

        # generate the note's samples
        for t in range(period):
            if (t < period / 2):
                samples[t] = amplitude
            else:
                samples[t] = -amplitude

        return samples

# waits until a note is pressed
def wait_for_note_start():
    while (True):
        for key in range(len(keys)):
            if (GPIO.input(keys[key])):
                return key
        sleep(0.01)

```

```

# waits until a note is released
def wait_for_note_stop(key):
    while (GPIO.input(key)):
        sleep(0.1)

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
    MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []

# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)

# create the actual notes
for n in range(len(freqs)):
    notes.append(Note(freqs[n], 1))

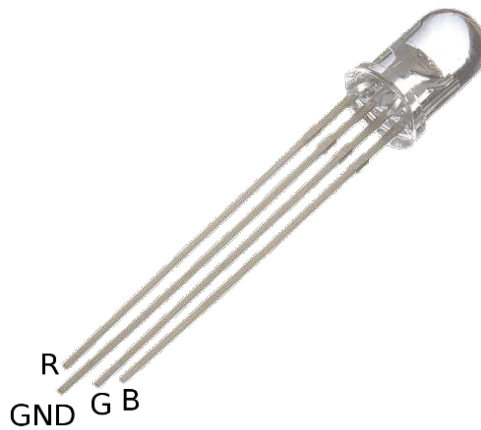
# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
    while (True):
        # play a note when pressed...until released
        key = wait_for_note_start()
        notes[key].play(-1)
        wait_for_note_stop(keys[key])
        notes[key].stop()
except KeyboardInterrupt:
    # reset the GPIO pins
    GPIO.cleanup()

```

Part three: recording and playback

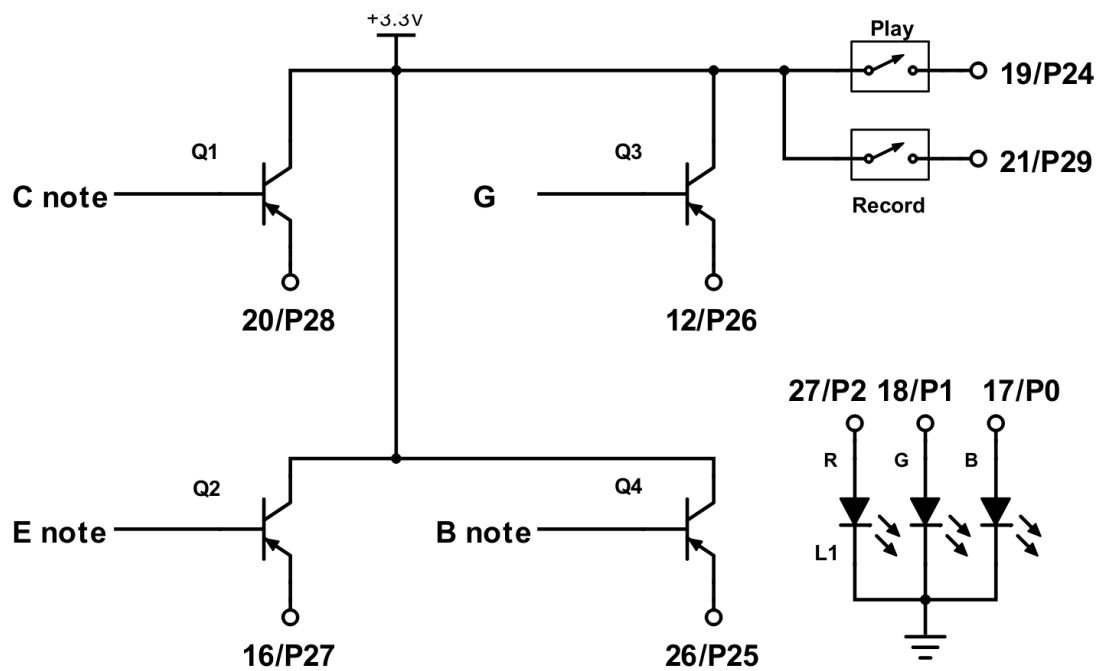
In this part of the activity, you will implement a record and playback feature. Two pushbuttons will be added (one for recording and one for playback). In addition, a RGB LED will be used to provide cues when the paper piano is recording (red LED) and playing back (green LED). Although two separate LEDs could be used for this, using the RGB LED will introduce you to the RGB LED. Let's take a closer look at the RGB LED:



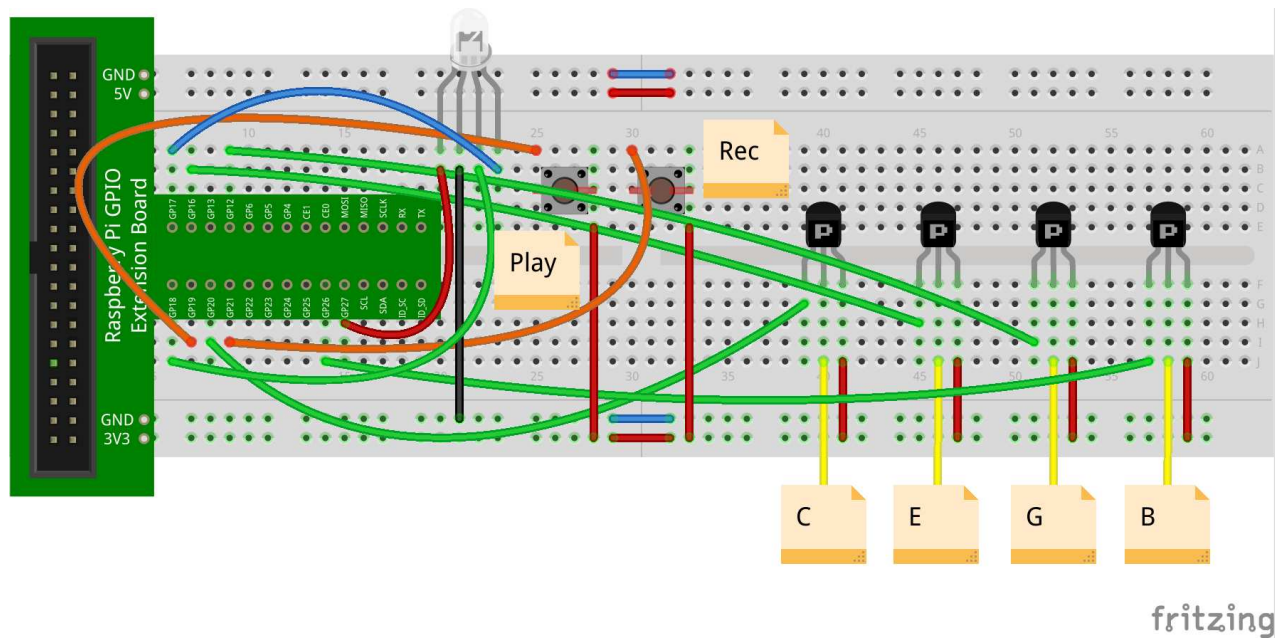
If oriented such that the longest leg is the second-from-the-left, then the legs in order from left-to-right are: red, ground, green, and blue. RGB LEDs that share a common ground are called common cathode RGB LEDs. This makes sense, since the ground leg of the LED is the cathode. As with the transistor, make sure that each leg of the LED is in a different column when inserting it into the breadboard. As expected, the ground leg is connected to ground. The three other (color) legs are connected to output pins on the RPi.

You will also note that there is no resistor on the RGB LED. Although one could be put in series with the ground leg, it really isn't needed since the RPi will only be supplying 3.3V. This is fine when turning on any of the color legs. You may notice that the red color LED is a bit dim when compared to green and blue. Although the activity uses the red LED to indicate that the paper piano is recording, you may instead use blue if desired. Green will be used to indicate playback.

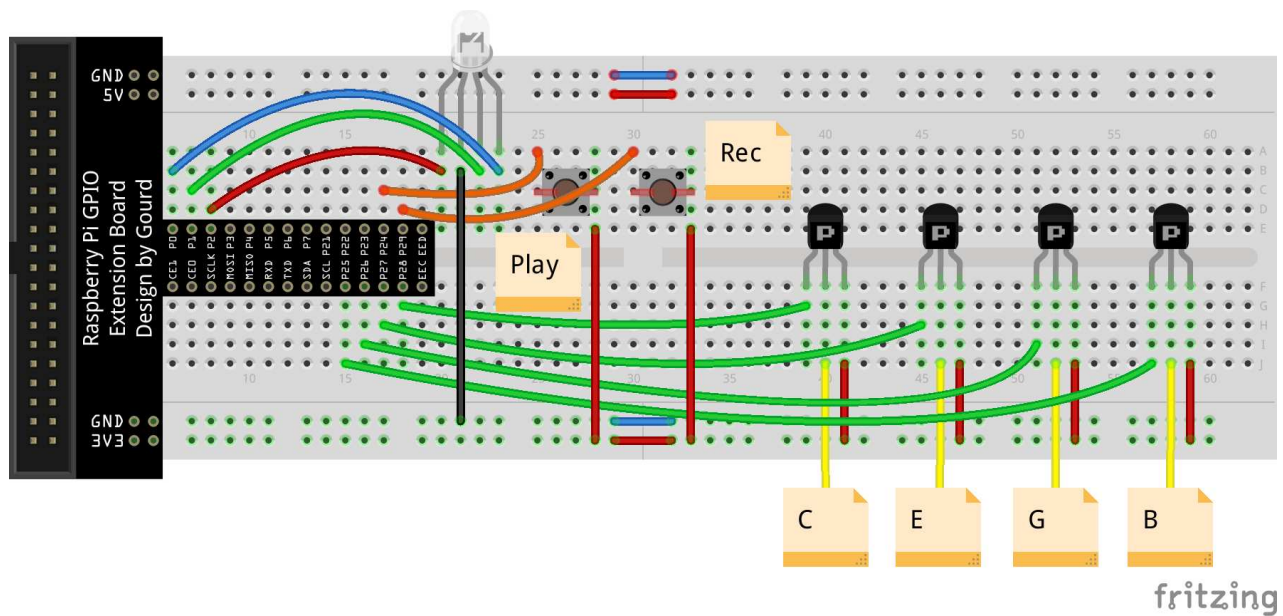
Here's the updated circuit:



Here's one way to layout this circuit:



If you have the black GPIO interface board, layout the circuit as follows instead:



Note that the right portion of the circuit with the transistors is the same as in the previous part of the activity. Also note that the play and record labels on the diagram simply serve to identify them. That is, there is no need to make labels with graphite on them since we are using pushbutton switches (and not capacitive touch switches) for record and playback functionality.

For reference, the play button is wired to GP19 (P24), and the record button is wired to GP21 (P29). The pushbuttons will also be wired to +3.3V; therefore, the input pins on the RPi will be configured so that they are pulled down (to ground) by default. The RGB LED colors are wired as follows: red to GP27 (P2), green to GP18 (P1), and blue to GP17 (P0).

The source code will need pretty major changes. First, detecting input now includes the pushbutton switches for recording and playback. Of course, these trigger the RGB LED as well. In addition, we need to support recording as the user plays the paper piano. This means that both the notes played and their duration (i.e., how long the notes were held down) must be stored. In addition, the duration of any silence in between the notes played must also be recorded! Playback of the recorded “song” must be able to play the notes and their duration, and any silence in between the notes.

We will structure the behavior of the recording and playback functionality so that the record button starts and stops recording. That is, if the program is not currently in a recording state, pressing the record button will put it in that state and turn on the red color LED. Pressing the record button again will toggle the recording state (to off) and turn off the LED. Pressing the play button will turn on the green color LED and play the recorded song. If the program is in a recording state, the play button will also stop recording and trigger the recording state (to off), turn the green color LED on, and play the song. That is, the play button will also stop recording if the program is in the recording state.

Let's take a look at the code. First, to implement timing (e.g., to record the duration of notes and silences), we import the time function from the time library:

Replace:

```
from time import sleep
```

With:

```
from time import sleep, time
```

The next change occurs in the `wait_for_note_start` function. We need to be able to detect when the play and record buttons are triggered, in addition to the notes:

Replace:

```
def wait_for_note_start():
    while (True):
        for key in range(len(keys)):
            if (GPIO.input(keys[key])):
                return key
        sleep(0.01)
```

With:

```
def wait_for_note_start():
    while (True):
        # first, check for notes
        for key in range(len(keys)):
            if (GPIO.input(keys[key])):
                return key
        # next, check for the play button
        if (GPIO.input(play)):
            # debounce the switch
            while (GPIO.input(play)):
                sleep(0.01)
            return "play"
        # finally, check for the record button
        if (GPIO.input(record)):
            # debounce the switch
            while (GPIO.input(record)):
                sleep(0.01)
            return "record"
        sleep(0.01)
```

The strategy is to first check if a note is being played. If so, its key is returned. If not, we then check if the play button is being pressed. If so, we debounce it by waiting until it is no longer pressed; then, we return the string “play.” If the play button is not being pressed, then we check if the record button is being pressed. If so, it is handled similarly to the play button; however, the returned string is “record.” If no button or note is being pressed, the program sleeps for a short while and checks again.

Next, we add the setup for the new GPIO pins (both for recording and playback, and for the RGB LED):

Replace:

```
# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []
```

```
# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)

# create the actual notes
for n in range(len(freqs)):
    notes.append(Note(freqs[n], 1))
```

With:

```
# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []
```

```
# setup the button pins
play = 19
record = 21
```

```
# setup the LED pins
red = 27
green = 18
blue = 17 # if red is too dim, use blue
```

```
# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(play, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(record, GPIO.IN, GPIO.PUD_DOWN)
```

```
# setup the output pins
GPIO.setup(red, GPIO.OUT)
GPIO.setup(green, GPIO.OUT)
GPIO.setup(blue, GPIO.OUT)
```

```
# create the actual notes
for n in range(len(freqs)):
    notes.append(Note(freqs[n], 1))
```

Note that no actual code was replaced or removed (i.e., only new statements were added). First, we add the input pin definitions for the record and play buttons. Then, we add the output pin definitions for the RGB LED. Lastly, we add the GPIO setup code for the new input and output pins.

Then, we initialize a list that will contain the recorded song. We also initialize the recording state to false. This is done before the actual start of the main part of the program:

Replace:

```
# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."
```

With:

```
# begin in a non-recording state and initialize the song
recording = False
song = []

# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."
```

Each element in the song list will contain both the note played (or a silence), along with its duration. These will both be stored as a list. So, the song list is a list of lists. Here's an example of one that contains the note C for 3.5s, a silence for 1.2s, and the note G for 2.45s:

```
[ [ "C", 3.5 ], [ "SILENCE", 1.2 ], [ "G", 2.45 ] ]
```

Next, we modify the loop in the main part of the program:

```
Replace;
try:
    while (True):
        # play a note when pressed...until released
        key = wait_for_note_start()
        notes[key].play(-1)
        wait_for_note_stop(keys[key])
        notes[key].stop()
except KeyboardInterrupt:
```

With:

```
try:
    while (True):
        # start a timer
        start = time()
        # play a note when pressed...until released (also
        # detect play/record)
        key = wait_for_note_start()
        # note the duration of the silence
        duration = time() - start
        # if recording, append the duration of the silence
        if (recording):
            song.append(["SILENCE", duration])
        # if the record button was pressed
        if (key == "record"):
            # if not previously recording, reset the song
            if (not recording):
                song = []
            # note the recording state and turn on the red LED
            recording = not recording
            GPIO.output(red, recording)
        # if the play button was pressed
        elif (key == "play"):
            # if recording, stop
```

```

        if (recording):
            recording = False
            GPIO.output(red, False)
            # turn on the green LED
            GPIO.output(green, True)
            # play the song
            play_song()
            GPIO.output(green, False)
        # otherwise, a piano key was pressed
    else:
        # start the timer and play the note
        start = time()
        notes[key].play(-1)
        wait_for_note_stop(keys[key])
        notes[key].stop()
        # once the note is released, stop the timer
        duration = time() - start
        # if recording, append the note and its duration
        if (recording):
            song.append([key, duration])
except KeyboardInterrupt:

```

There's a lot going on here. First, we start a timer to record any silence while waiting for a button to be pushed or note to be pressed. Once any input is received, the duration of the silence is stored. If the program is in a recording state, the silence and its duration is added to the song. If not, it is ignored.

Next, we determine what button was pressed or note was pushed. If the record button was pressed, we then check if the program is not in a recording state. If that is the case, then the song is initialized (or possibly reinitialized if a song already existed). The recording state is then toggled, and the red color LED is turned on. The loop then iterates again, recording the silence and waiting for a button to be pushed or a note to be pressed.

If the play button is pressed, we first check if the program is in a recording state. If so, the recording state is toggled, and the red color LED is turned off. Next, the green color LED is turned on, and the song is played (via a `play_song` function). Once the song has played, the green color LED is turned off. The loop then iterates again, recording the silence and waiting for a button to be pushed or a note to be pressed.

Lastly, the else part of the main if statement detects if a note is pressed. If so, the a timer is started to record the length of the note as it is played. The note is then played while the program waits for it to be released. Once this happens, the note stops playing, and its duration is stored. Lastly, if the program is in a recording state, the note and its duration are added to the song. The loop then iterates again, recording the silence and waiting for a button to be pushed or a note to be pressed. Of course, the program stops when the user presses Ctrl+C.

The last addition to the program is the `play_song` function. It can be placed below the `wait_for_note_stop` function:

```

# plays a recorded song
def play_song():
    # each element in the song list is a list composed of two
    # parts: a note (or silence) and a duration
    for part in song:
        note, duration = part
        # if it's a silence, delay for its duration
        if (note == "SILENCE"):
            sleep(duration)
        # otherwise, play the note for its duration
        else:
            notes[note].play(-1)
            sleep(duration)
            notes[note].stop()

```

Since the song list is initialized (and modified) in the main part of the program, it can be directly accessed in the `play_song` function. The function first iterates through each list in the song list. Remember that the song list contains lists! Each of the “sublists” contains either a note or a silence, and its duration.

The function breaks each part of the song into its note (or silence) and duration. If the part is a silence (i.e., it contains the string “SILENCE”), the program sleeps for its duration. Otherwise, the part contains a note that is played for its duration, and then stopped.

Once the entire song has been played, control is transferred back to the main part of the program, at which point the green color LED is turned off, and the program waits for another button to be pressed or note to be pushed.

For completeness, the entire modified source code is listed here:

```

#####
# Name:
# Date:
# Description: Paper piano (v3).
#####
import RPi.GPIO as GPIO
from time import sleep, time
import pygame
from array import array

MIXER_FREQ = 44100
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
    # note that volume ranges from 0.0 to 1.0

```

```

def __init__(self, frequency, volume):
    self.frequency = frequency
    # initialize the note using an array of samples
    pygame.mixer.Sound.__init__(self,\
        buffer=self.build_samples())
    self.set_volume(volume)

# builds an array of samples for the current note
def build_samples(self):
    # calculate the period and amplitude of the note's wave
    period = int(round(MIXER_FREQ / self.frequency))
    amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
    # initialize the note's samples (using an array of
    # signed 16-bit "shorts")
    samples = array("h", [0] * period)

    # generate the note's samples
    for t in range(period):
        if (t < period / 2):
            samples[t] = amplitude
        else:
            samples[t] = -amplitude

    return samples

# waits until a note is pressed
def wait_for_note_start():
    while (True):
        # first, check for notes
        for key in range(len(keys)):
            if (GPIO.input(keys[key])):
                return key
        # next, check for the play button
        if (GPIO.input(play)):
            # debounce the switch
            while (GPIO.input(play)):
                sleep(0.01)
            return "play"
        # finally, check for the record button
        if (GPIO.input(record)):
            # debounce the switch
            while (GPIO.input(record)):
                sleep(0.01)
            return "record"
        sleep(0.01)

# waits until a note is released
def wait_for_note_stop(key):

```



```

    while (GPIO.input(key)):
        sleep(0.1)

# plays a recorded song
def play_song():
    # each element in the song list is a list composed of two
    # parts: a note (or silence) and a duration
    for part in song:
        note, duration = part
        # if it's a silence, delay for its duration
        if (note == "SILENCE"):
            sleep(duration)
        # otherwise, play the note for its duration
        else:
            notes[note].play(-1)
            sleep(duration)
            notes[note].stop()

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
    MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []

# setup the button pins
play = 19
record = 21

# setup the LED pins
red = 27
green = 18
blue = 17 # if red is too dim, use blue

# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(play, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(record, GPIO.IN, GPIO.PUD_DOWN)

# setup the output pins

```

```

GPIO.setup(red, GPIO.OUT)
GPIO.setup(green, GPIO.OUT)
GPIO.setup(blue, GPIO.OUT)

# create the actual notes
for n in range(len(freqs)):
    notes.append(Note(freqs[n], 1))

# begin in a non-recording state and initialize the song
recording = False
song = []

# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
    while (True):
        # start a timer
        start = time()
        # play a note when pressed...until released (also
        # detect play/record)
        key = wait_for_note_start()
        # note the duration of the silence
        duration = time() - start
        # if recording, append the duration of the silence
        if (recording):
            song.append(["SILENCE", duration])
        # if the record button was pressed
        if (key == "record"):
            # if not previously recording, reset the song
            if (not recording):
                song = []
            # note the recording state and turn on the red LED
            recording = not recording
            GPIO.output(red, recording)
        # if the play button was pressed
        elif (key == "play"):
            # if recording, stop
            if (recording):
                recording = False
            # turn on the green LED
            GPIO.output(red, False)
            GPIO.output(green, True)
            # play the song
            play_song()

```

```

        GPIO.output(green, False)
    # otherwise, a piano key was pressed
    else:
        # start the timer and play the note
        start = time()
        notes[key].play(-1)
        wait_for_note_stop(keys[key])
        notes[key].stop()
        # once the note is released, stop the timer
        duration = time() - start
        # if recording, append the note and its duration
        if (recording):
            song.append([key, duration])
except KeyboardInterrupt:
    # reset the GPIO pins
    GPIO.cleanup()

```

Make sure to test the program with several runs of different behaviors. Try recording a few songs, some that don't have notes, some with different silences and durations, and so on.

Homework: Paper Piano

An interesting tweak to this activity is to change the shape of the wave for each note as generated by the mixer. That is, keep the frequency and sample rate constant, but change the amplitudes across the samples so that the shape of the wave changes. As noted earlier, the current program generates square waves. This is done in the `build_samples` function of the `Note` class:

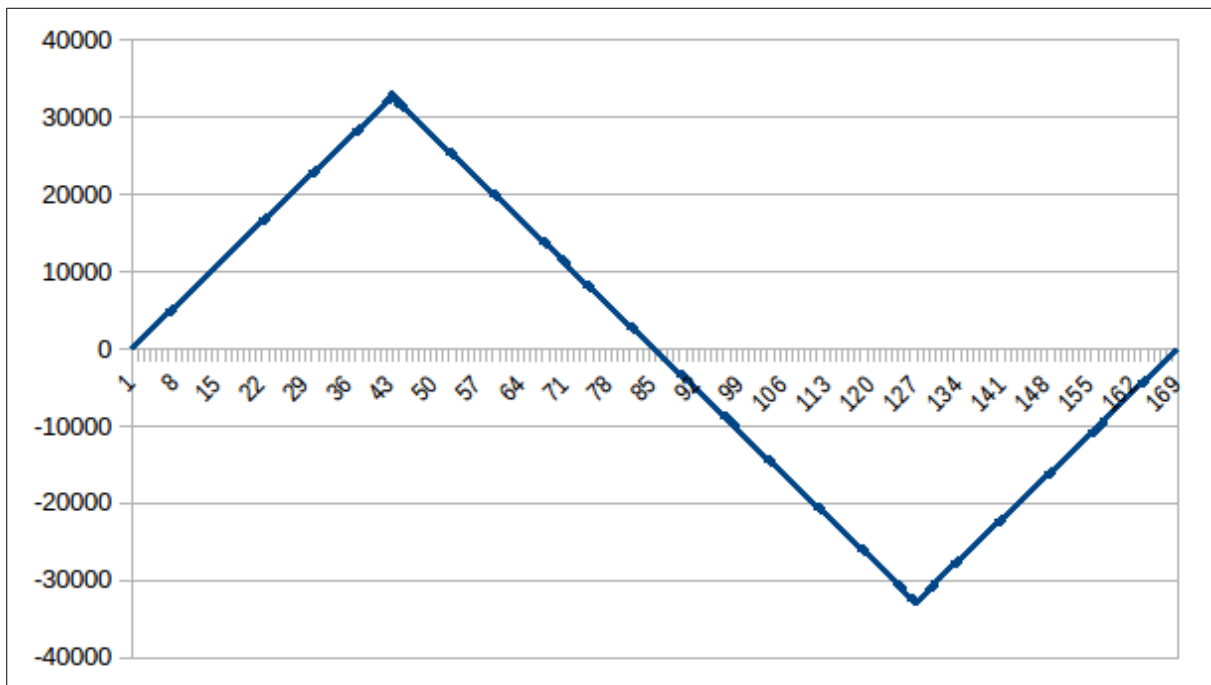
```

# generate the note's samples
for t in range(period):
    if (t < period / 2):
        samples[t] = amplitude
    else:
        samples[t] = -amplitude

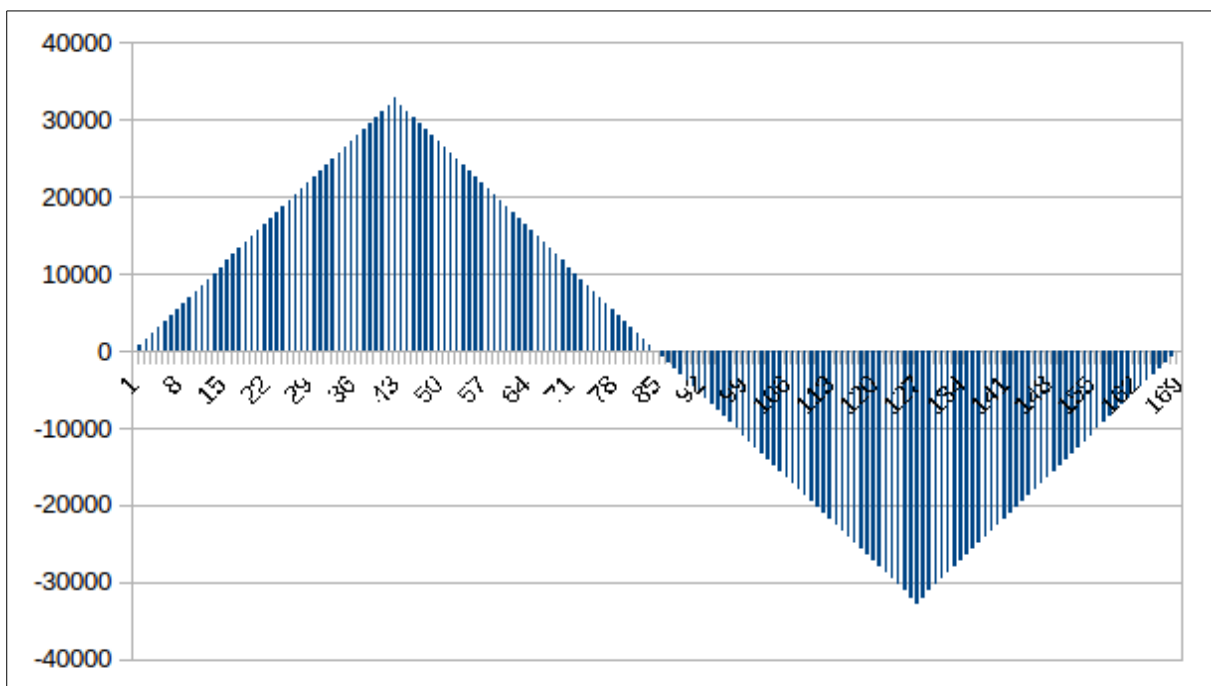
```

Note that half of the samples are at the maximum amplitude, and half are at the negative of the maximum amplitude.

There are other kinds of waves (or wave approximations) that can be generated, however. Consider a modification that forms a **triangle wave**:



We could, instead, fill the array of samples so that it forms a triangle wave as follows:

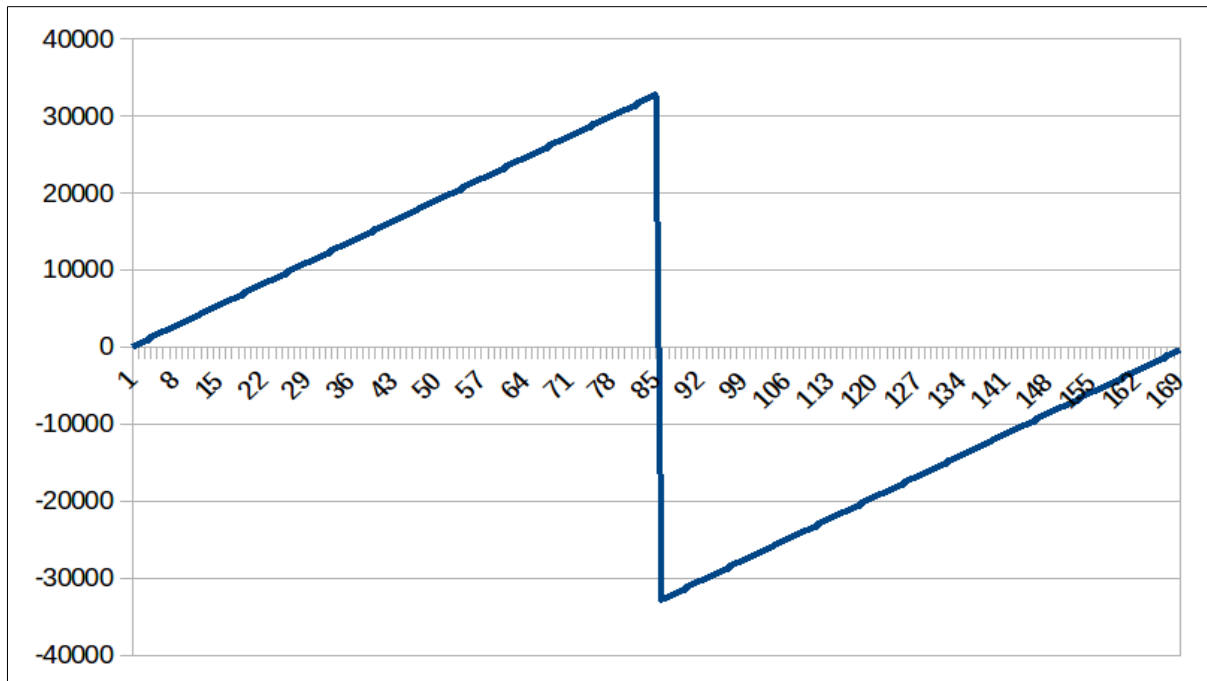


The modification to the source code snippet shown above is fairly simple. Algorithmically, we simply need to increase the magnitude of each sample from 0 to 32,767 (the maximum amplitude, also the maximum value for 16-bit signed numbers) until we reach one fourth of the total number of samples. For a middle C at 261.6 Hz with 169 total samples in a period within a 44.1 KHz sample rate, that's about 42 samples. The increase (or delta) at each sample can easily be calculated. Then, decrease the magnitude of each sample from 32,767 to -32,768 (the smallest value for 16-bit signed numbers)

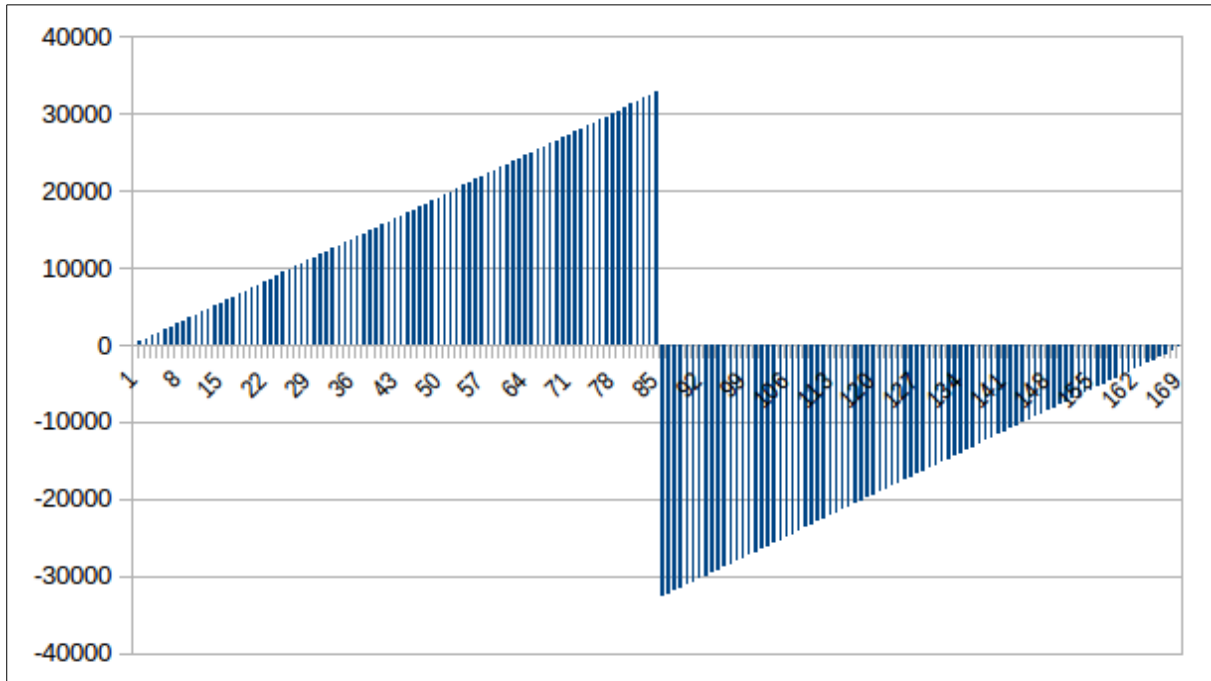
through the next half of the samples (i.e., through three quarters of the total number of samples – about 85 samples). Finally, increase the magnitude of each sample from -32,768 to 0 until we reach the end of the total number of samples.

What does such a sound wave sound like?

What about a wave that looks jagged (like saw teeth). Such a wave is called a **sawtooth wave**:

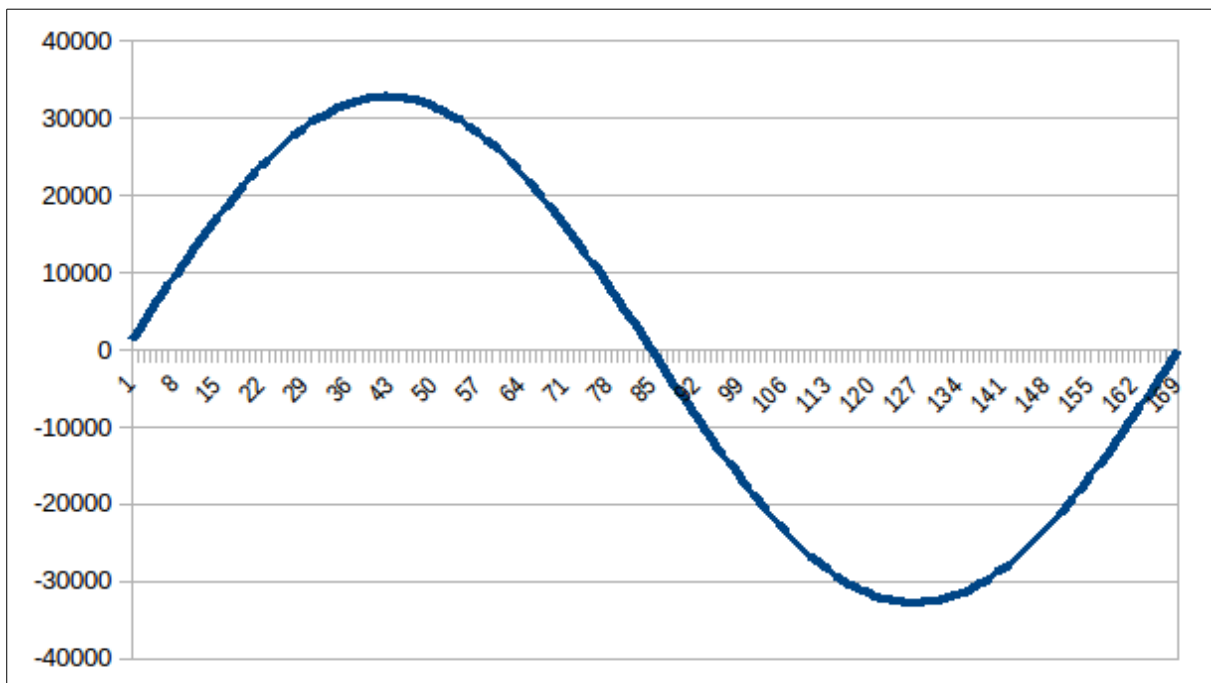


Similarly, we could fill the array of samples so that it forms a sawtooth wave as follows:

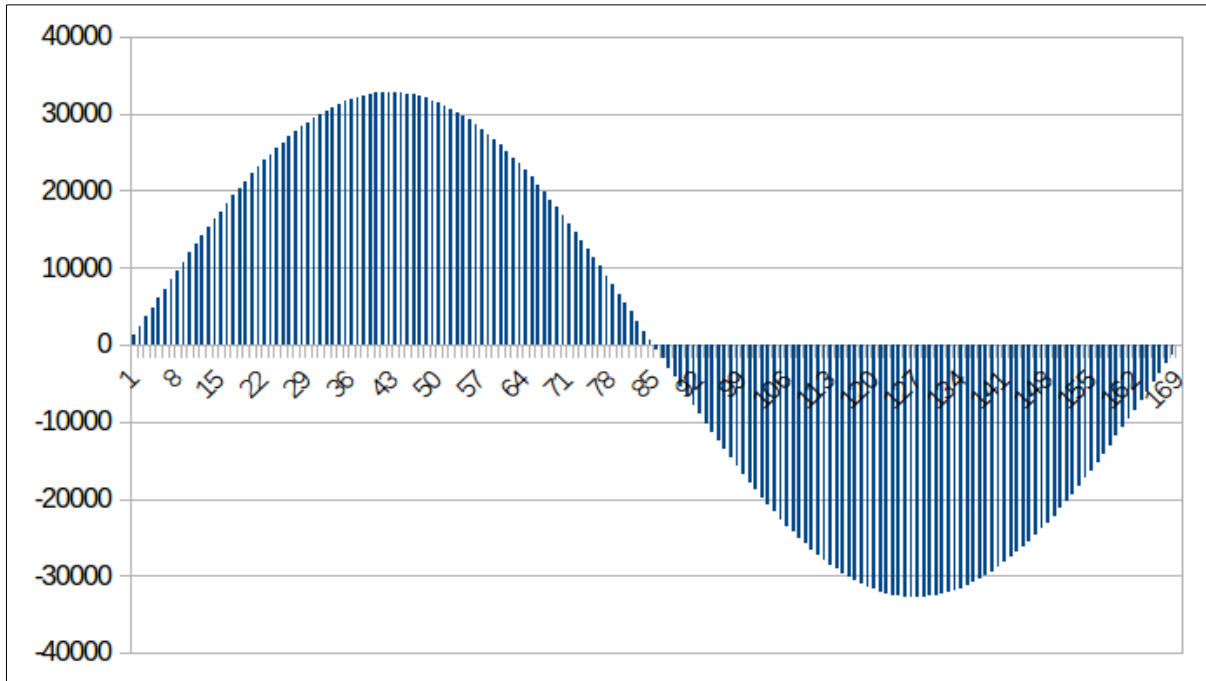


The modification to the snippet of code for this type of wave is also fairly simple. How different does this type of wave sound as compared to the others?

Finally, perhaps the most difficult set of samples to generate are those that represent the **sinusoidal wave** as accurately as possible:



Again, we could fill the array of samples so that it forms a sinusoidal wave as follows:



How different does this type of wave sound as compared to the others? Although this is possible by modifying the snippet of code above, it will require more changes than the other types of waves. In fact, various trigonometric functions will be necessary.

Your task is to modify the Paper Piano to implement **at least two** of the types of waves shown above (triangular, sawtooth, and sinusoidal) **in addition to the existing square wave**. **Implementing all three may result in bonus points!**

The behavior of the Paper Piano will change slightly to accommodate the changes specified. Each of the note inputs must be set to the same note (and same frequency). Therefore, the piano no longer plays C, E, G, and B. Instead, it plays a single note (**let's agree to use a middle C at 261.6 Hz**); however, it does so using different wave forms: pressing one of the keys plays a middle C using a square wave, pressing another key plays a middle C using another wave form of your choosing (e.g., sawtooth), and so on. If you implement two of the additional wave forms, then your piano will only require three keys. Implementing all three will require all four keys.

Efficient solutions to adding the various wave forms will include a modification of the Note class to accept a wave form type upon instantiation. Sample building will then depend on type of the wave form for the current note object.

Although you may remove the play and record buttons (and their functionality), this is not necessary. In fact, it may be interesting to record and playback the differences in the wave forms. **If you choose to remove the play and record buttons, it will be much easier if you begin with the source code specified in part two of this activity.**

Include some comments in your source code that briefly discuss the differences in how the different wave forms for a middle C sound to you.

You may have the option to work in **groups** (pending prof approval). It is suggested that groups contain at least one confident Python coder. Make sure to put an appropriate header at the top of your program (with the names of everyone in your group, if allowed and applicable) and to appropriately comment your source code as necessary. **Only submit your source code (i.e., a single .py file).**

Raspberry Pi Final Project

This is the final RPi activity (in fact, it's your final project). You may consider it more than just an activity, as it should (in theory) attempt to encompass everything (or as much of everything as possible) that you have learned in the Living *with* Cyber curriculum. Your project should contain both software and hardware combined in such a manner as to satisfy the theme of the project. It goes without saying that it should run on a Raspberry Pi.

The project is somewhat open-ended, in that you have some degree of freedom in terms of the project's topic, focus, and goals. However, your project must be approved by the prof!

Let's talk specifics:

- (1) You will work **in groups of three**;
- (2) You will have class time to brainstorm and work on your project;
- (3) Any source code must be written in Python, unless approval otherwise is obtained from the prof;
- (4) Your project may, in some way, make use of GPIO. That is, it can include input(s) (i.e., from sensors) and produce output(s) that respond to the input(s). Note that the definition of inputs and outputs is somewhat flexible;
- (5) Your project must incorporate an intuitive GUI that utilizes the LCD touchscreen, unless approval otherwise is obtained from the prof;
- (6) You must use github as the main repository for your project's source code and other files; and
- (7) Your project will be evaluated on the following:
 - (i) Complexity: does your code reflect the work of students who have been programming for a whole year?
 - (ii) Efficiency: is your code wasteful with respect to space or time?
 - (iii) Data structures: does your code make use of data structures (if applicable)?
 - (iv) User interaction: is the GUI intuitive?
 - (v) Robustness: is your code resistant to unexpected user input?
 - (vi) Readability: can someone from another group (and/or the prof) read and understand your code easily?
 - (vii) I/O (if applicable): does your project leverage the benefits of GPIO that the RPi offers?

Deliverables include:

- (1) Initially (**this will be delivered at the beginning of the project and must be approved by the prof before you actually begin working on your project!**):
 - (i) A brief writeup of your project that summarizes it (note that you must include justification that your idea can be translated into a project that can be completed by a group of three-ish students in the allotted time).
- (2) At the conclusion of the project:
 - (i) All source code;
 - (ii) If using GPIO and a circuit, both a circuit diagram (schematic) and layout diagram (**done in Fritzing**);
 - (iii) A BOM (bill of materials; i.e., a parts list), if applicable;
 - (iv) A link to your github repository for the project; and
 - (v) A final presentation of the project.

Note that all files (source code, images, circuits, etc) should be compressed **in a single ZIP file** (not RAR, etc).

The final presentation should be approximately **10 minutes per group**. Everyone in your group should speak during the presentation. It should include the following:

- (1) A good overview of the project (i.e., what its goals are, how it works, why it is relevant, etc);
- (2) A demonstration of your project; and
- (3) Any future development plans and lessons learned.

If you wish, you can choose to make a video to be shown during your final presentation. If you choose to make a video, please adhere to the following requirements:

- (1) It must be a Youtube video;
- (2) The length of the video should be **between two to five minutes**; and
- (3) It should have a description and goals of the project.

Note that the video is only a part of your presentation (i.e., it shouldn't replace your presentation). If you choose to make a video, be creative!

Raspberry Pi Final Project **EVAULATION**

I am a: _____Student _____Faculty Other (specify):_____

Group members: _____

Project title: _____

Organization and clarity **5 pts** _____
 (1) Is the presentation organized in a logical manner?

Technical content **65 pts** _____
 (1) Is the difficulty level appropriate (i.e., was the project nontrivial)?
 (2) Is the material presented technically sound?
 (3) Is the project relevant to the specified topic(s)?
 (4) Does the project include a discussion of hardware and software components as applicable, future development plans, and lessons learned?

Timing **5 pts** _____
 (1) Was a 10 minute presentation delivered (+/- 1 minute)?

Demonstration **10 pts** _____
 (1) Did the demonstration work?

Delivery **15 pts** _____
 (1) Do the presenters make eye contact with the (entire) audience?
 (2) Are the presenters expressive in voice, posture, and movements?
 (3) Did all group members participate in the presentation (i.e., did they all speak)?

TOTAL **100 pts** _____