CSC/CYEN 130

The Science of Computing I

Living *with* Cyber

Student Edition

CSC/CYEN 130: The Science of Computing I
Living *with* Cyber (part 1 of 3)

Course Description: An introduction to computing, algorithm analysis and development, computer programming, data structures, computer architecture, and problem solving. This is the first Living *with* Cyber course.

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

1. Be able to identify a problem's variables, constraints, and objectives;
2. Be able to represent algorithms in various ways (e.g., flowcharts, pseudocode);
3. Have a basic understanding of algorithms (e.g., searching and sorting) and their complexity;
4. Be able to write simple programs in a general purpose programming language (e.g., Python);
5. Have a basic understanding of introductory data structures (e.g., arrays);
6. Have a basic understanding of the functional components of a computer (e.g., CPU, memory);
7. Understand logical operations (e.g., AND, OR, and NOT) on binary inputs and their translation to digital gates; and
8. Have an understanding of computing as it applies in and how it affects the global context.

Prerequisite(s): A grade of **C** or better in MATH 101 or equivalent.

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:	~5%
Puzzles:	~2.5%
Raspberry Pi activities:	~22.5%
Programs:	~17.5%
Other assignments:	~2.5%
Major tests:	~50%

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

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

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

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

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

TOPICS COVERED:

- Origin
- Lecture 0
- Introduction to Living *with* Cyber
- Introduction to Algorithms
- Introduction to Computer Programming
- Introduction to Computer Architecture
- Searching and Sorting
- Introduction to Data Structures

Lesson Summary

#	Title	Pillar(s)	Description/Topic(s)	Periods
00	Origin	Foundation	1. Summary of the course 2. Housekeeping	0.5
01	Lecture 0	Foundation	1. Philosophical underpinnings of Living *with* Cyber 2. Bag of tricks (to help with problem solving)	0.5
02	Introduction to Living *with* Cyber	Foundation	1. What is Living *with* Cyber? 2. What is cyber? 3. What is computer science?	1
03	Introduction to Algorithms	Algorithms	1. Introduction to algorithms 2. Problem statements 3. Step breakdown and control flow 4. To-do lists and flowcharts 5. Repetition in algorithms 6. Efficiency and runtime analysis 7. Computer programs and pseudocode	2
04	Introduction to Computer Programming	Computer Programming	1. Machine language, programming language, compilation, interpretation 2. Programming paradigms 3. Introduction to the Python programming language 4. The IDLE IDE 5. Scratch vs. Python 6. Python primer 7. Data types, constants, variables, I/O 8. Expressions and assignment 9. Subprograms 10. Operators 11. Identifiers and reserved words 12. Comments 13. Primary control constructs (sequence, selection, repetition) 14. Recursion 15. Program flow	5
05	Introduction to Computer Architecture	Computer Architecture	1. Introduction to computer architecture 2. Fundamentals of digital logic 3. Circuit and layout diagrams 4. Ohm's Law 5. Logic gates (*and*, *or*, and *not*) and Boolean algebra 6. Combinational circuits (*xor* and various comparators)	3

06	Searching and Sorting	Algorithms	1. Searching (covers the sequential search) 2. Sorted searching (covers the binary search) 3. Sorting (covers the bubble sort, selection sort, and insertion sort) 4. Sort comparisons and efficiency analysis	3
07	Introduction to Data Structures	Data Structures	1. Introduction to data structures 2. 1D arrays 3. The Python sequence and Python lists 4. Creating and populating an array 5. The Python for loop 6. Performing a sequential search on an array 7. Performing a selection sort using an array 8. Performing a binary search on an array	3
Pi Activities				7
Exams				3
Slack				2
TOTAL				**30**

CSC/CYEN 130: The Science of Computing I

Last updated:

13 Mar 2018

		Lessons	Raspberry Pi Activities	Puzzles	Videos	Assessments	
WEEK 1	1	Origin -- Lecture 0			Why do we Ignore Failure? (parts 1 and 2)		F
	2	Introduction to Living *with* Cyber				Paper Future of Computing	M
	3		Pi Activity 1 Sampling Some Raspberry Pi				W
WEEK 2	4	Introduction to Algorithms		Mystery Mania	You can Learn Anything		F
	5						M
	6		Pi Activity 2 *Scratch*ing the Surface				W
WEEK 3	7						F
	8	Introduction to Computer Programming		Count the Squares -- Cutting the Rug	The Next Big Idea in Interactive Gaming		M
	9					Program 1 Easy Does It	W
WEEK 4	10						F
	11					Program 2 Easy Does It...Reloaded	M
	12						W
WEEK 5	13					Exam 1 Algorithms Computer Programming	F
	14	Introduction to Computer Architecture		Ten Characters, 13 Boxes -- 8 Boxes		Program 3 Casual Statistics	M
	15						W

CSC/CYEN 130: The Science of Computing I

Last updated:

13 Mar 2018

		Lessons	Raspberry Pi Activities	Puzzles	Videos	Assessments	
WEEK 6	16					Program 4 99 Bottles of Beer on the Wall	F
	17		Pi Activity 3 LED the Way				M
	18						W
WEEK 7	19					Exam 2 Computer Architecture	F
	20	Searching and Sorting					M
	21			Wolf, Goat, Cabbage -- Four Men and a Torch		Program 5 Search Comparison Table	W
WEEK 8	22				15 Sorting Algorithms in 6 Minutes		F
	23	Introduction to Data Structures				Program 6 A Fantastically Fabulous List	M
	24						W
WEEK 9	25			Connect 1, 2, 3 -- Symbol Sequence	How a 5'5" Basketball Player Learned to Dunk	Program 7 Random List Statistics	F
	26					Exam 3 Searching and Sorting Data Structures	M
	27		Pi Activity 4 Ultrasonic Sort				W
WEEK 10	28						F
	29	SLACK					M
	30						W

Lessons

The Raspberry Pi platform
Living *with* Cyber makes use of a unique computing platform that you will use throughout the curriculum. It consists of a Raspberry Pi computer (yes, it's a computer), an LCD touchscreen, a keyboard and mouse, USB-powered speakers, and various other parts. With this platform, you will complete a variety of hands-on activities. We will try to tie them all to *cyber* in some way (this is why we call this curriculum Living *with* Cyber). We'll explain what *cyber* is later.

Puzzle-based learning
The curriculum also makes use of puzzle-based learning. The idea behind this learning approach is to utilize puzzles to cultivate problem solving and critical thinking skills. Puzzles are fun, and if we can think of large, complicated problems as puzzles, then perhaps we can actually have fun solving them. Throughout the curriculum, we will discuss (and solve) various puzzles. By doing this, we will try to identify certain tactics that can be used to approach problem solving. You will also be assigned some to do on your own. It is understandable that, particularly at first, you may have a hard time solving the puzzles. We hope that, over time, you will become better at solving them.

Videos
The curriculum also makes use of videos that are shown in class. They are intended to be thought-provoking and cover a wide range of topics. Some relate to interesting applications of computer science; others cover more philosophical points about learning.

Rules when taking exams
Here are a few rules to keep in mind when taking exams:

1. *No cheating*. It's sad that this has to be mentioned; however, it does occur. At the least, you will earn an F on the exam. At the most, you will earn an F in the course and will be recommended for expulsion from the university.
2. *Cell phones must be turned off* . They can't even be used as a watch.
3. *No reference material can be used*. Exams are closed book and closed notes.
4. *No sharing of anything*. Work the exam on your own. This is your time to show what you (and only you) have learned and can do.

Completing homework and exams
Here are a few suggestions when completing homework assignments and exams:

1. *Be clear and concise*. Don't vomit words on the page.
2. *Write legibly*. If the prof can't read your writing, it will most likely be marked wrong.
3. *Doodle to elaborate*. This is useful to clarify your words. It also helps you to problem solve.
4. *Format source code properly*. On homework, consider taking screenshots of source code.

Proper email etiquette
The art of *good emailing* is unfortunately not widespread. Here are a few things to keep in mind when drafting an email to your prof:

1. *Make your subjects short and meaningful*. We should be able easily prioritize it by the subject.
2. *Make your message short and meaningful*. Be concise but also clear. Please provide adequate detail to get your message across without inducing a puzzled look on your prof's face.
3. *Don't use* **ALL CAPS**. This is rude and quite annoying. It is usually taken as yelling.

4. *Write properly*. Use proper spelling, grammar, punctuation, and capitalization. Email is not texting.
5. *Sign your name*. Do this at the bottom of your message. Please. Sign. Your. Name. The prof can't (and shouldn't) try to figure out who you are by looking at your email address. Note that you can automatically sign each email by adding a signature.

Please make sure to check your @latech.edu email **daily**. There are times when your prof needs to communicate important things to you with urgency.

Make the most of your time in class

Here are a few things that can help you perform well in class and make it better for all of us:

1. *Listen*. Don't chitchat with your classmates. This can disrupt the prof and your classmates.
2. *Be active*. Take notes, ask questions, work class problems, and help others when allowed.
3. *Turn your cell phone off*. Or at least set it on vibrate. Don't answer it in class. If you must answer, please leave the room before doing so.
4. *Use laptops for class activities*. Please don't check Facebook, email, the news, play games, etc.
5. *Take responsibility for your area of the room*. Throw away trash around you, wipe off dirty tables, tuck your chair in, etc. Basically, just be a good human being.

Some not-so-obvious reminders

Although not directly related to this course and to the classroom, these nonetheless important tidbits come courtesy of Professor Paul Crook in the Department of Theatre:

1. Crosswalks are a thing;
2. Yes, pedestrians have the right of way; however, **you still have to obey traffic signs and signals**;
3. If a 2000+ pound vehicle is mere feet from your body, you might want to stop staring at your phone and move it along;
4. Sidewalks and stairwells are two-way streets, and I reserve the right to body-check you if you and your five friends are walking abreast – or if you are walking in the middle while staring at your phone and not paying attention to your surroundings;
5. Faculty parking is for, you know, faculty – it's not my fault, and I don't care that you woke up late for class; and
6. Grumpy Mondays are, well, grumpy...

One last thing

Often, students new to the university "life" find it hard to speak to their professors. Most of us are normal people just like you, and we're pretty much open to chatting about concerns that you may have about the course. It is to your advantage (greatly so, in fact) to use your prof's office hours wisely! If you have questions or need clarity on a problem, please see your prof during his or her office hours. If your prof's office hours are not convenient for you (e.g., you have other classes during that time), please communicate with your prof via email (using proper email etiquette, of course). We are usually open to setting up an appointment to accommodate you.

Lastly, if you do average work on every assignment, activity, exam, etc, **and submit them all on time**, it is likely that you will pass the course. Missing even a single assignment (or, even worse, an exam) will significantly impact your final grade in the course. Bottom line: **every assignment matters!** If you are absent (e.g., due to being sick or attending some sort of university sponsored activity), please make sure to communicate with your prof **in advance** (if you can) via email. Communication is really important.

On your marks...

What is computer science?

This is exactly the question that we will attempt to answer in this curriculum, and at its conclusion you should be able to answer that very question. But it's not a simple one. Computer science is a large topic that is composed of a great many things, of which one small part is coding (what most people think of when they hear computer science). We will try to explore as many of these things as we can. Fundamentally, though, computer science is about solving problems. The end goal of this curriculum is to understand what computer science is while simultaneously becoming better at solving problems (which, by the way, is something that we will need to do for the rest of our lives).

Coding is easy

That's what it seems like to a lot of people. In fact, this is a fallacy; however, the idea that it's easy (or that it should be) appears to be widespread. We see this on TV, in advertisements that try to get young people interested in computing, on the Internet, in social media, and so on. The problem with this is that people who start learning to code naturally assume that it will be easy for them as well. And then they hit hurdles.

Coding is a bit like learning to play a sport. Let's pick on racquetball since it's a great game that some people love to play. It takes a while at first to play well. It's not easy to hit a small blue ball that is moving across a small court very fast. By the way, the court has four walls and a ceiling – a lot of surfaces for that ball to bounce off of. After playing for some time, we become pretty good. We win many games and only lose some. We win enough to feel good about our performance. But then we hit a plateau. That plateau sometimes lasts for *years* (see Figure 1). It's not fun to play a game so much and not become any better for a long time. But if we play long enough, we do become better. If we play long enough. And if we don't quit. It's just how long it takes.

The same can be said with, for example, painting. Paintings often elicit emotion. We can look at a painting and admire its beauty. We can judge a painting and comment on whether it was done by an expert or a beginner. In fact, it's pretty easy to do so. An expert painter could spend one month with us to explain and demonstrate everything that there is to know about painting: the canvas, brushes, paint, the palette, mixing colors, brush techniques, etc.

And then it's our turn to paint. What would our painting look like, even after knowing a lot about painting? Most likely, not too good. We have the *knowledge*; what we lack is *understanding*. The solution is to practice, over and over again. We will fail many times, but eventually we will produce a painting that we are proud of! Coding is like this. It takes practice, often on our own. It takes more time and effort than many other things. It's just how long it takes.

Failure

Let's face it, quitting is easy. We often assume that if we fail at things, it must mean that we're not good at them. And for some reason, we also assume that we'll *never* be any good at them. So we quit, especially if something is supposed to be easy since everyone seems to think so. And if coding is not easy for us, then coding must not be for us. We must not be *built* to be a coder, and most likely our mind just *works* differently than those who are good at coding. Let's be honest, failure sucks.

Failure should be our teacher, not our undertaker. Failure is delay, not defeat. It is a temporary detour, not a dead end. Failure is something we can avoid only by saying nothing, doing nothing, and being nothing. – Denis Waitley

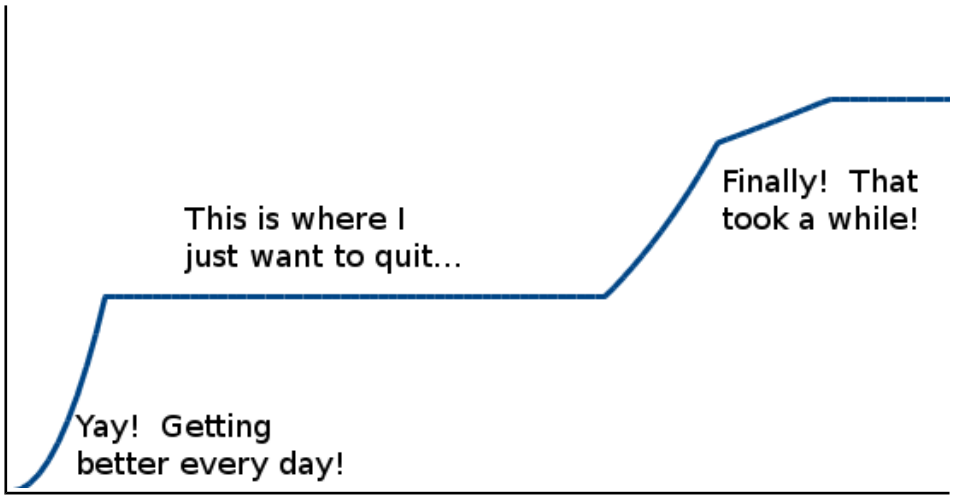
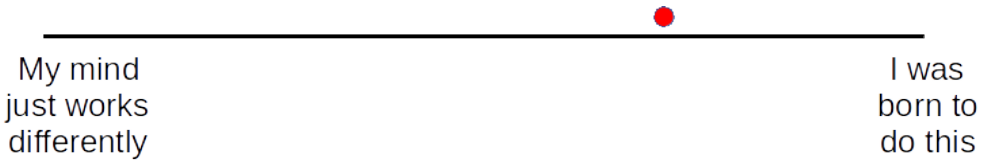
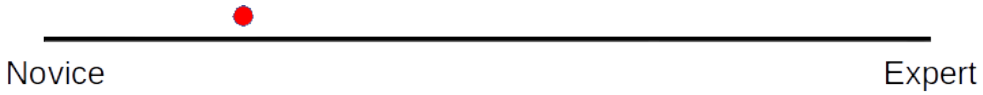


Figure 1: Racquetball learning curve. Horizontal axis is time spent playing; vertical axis is skill.

But the truth is that most programming doesn't require a special brain (or something that we're somehow *born* to do). There is no such thing as "I'll never be good at coding" or "I was born to code!" We can't simply plot our *ingrained* ability on a scale like this:



In reality, it's more like this:



This scale is more realistic and provides a more reasonable meaning to quantifying one's expertise in coding. We all begin as novices with no experience. By learning, and through practice, we become better and shift to the right on this scale. It is true, however, that we are all quite different: we have different abilities, skills, learning strategies, experiences, prejudices, developed behaviors, and so on. We may not all shift to the right at the same velocity. Some of us may make progress and accelerate quickly while others may need **more** practice to get to the same point. Some of us will reach a plateau that will last a very long time, while others may blast through it in much less time. And some of us may encounter more problems along the way than others. Welcome to the real world. Remember this (attributed to Theodore Isaac Rubin): The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.

Failure is no fun. It can be awful. But living so cautiously that you never fail is worse. – J.K. Rowling

Although the right side of the scale above is labeled **expert**, there is no such thing as an expert coder. There is no final level in coding that, once reached, means that it has somehow been mastered. Becoming better at coding is something that we will work on continuously from the time that we start coding to the time that we stop coding. It's almost as if someone is constantly moving the right end. Frustrating, isn't it?

Men vs. women

There is this perception that men have what it takes to do well in computer science and that women somehow do not. The truth is that research does *not* substantiate this. In fact, it is diametrically in opposition to this. It is true that men and women are different: we look different, we age differently, we have somewhat different lifespans, we have a different center of mass which affects how we walk, and so on. But it is not true that we are somehow born to have what it takes (or not) in terms of our potential to excel at coding – or in general, computer science.

However, it is much more frustrating and messy than those who do it well let on. In other words, the plateau is often at play. And when we hit the plateau, we may feel that we are not making any significant progress for quite some time. And that is just discouraging. But the thing to do is to keep trucking along, to keep coding and producing work. That's just how long it takes.

Success

Very early on we may have a pretty good idea of what good code looks like. We may be able to take a look at someone else's work and determine if and how well it solves some given problem. We may be able to comment on its beauty and efficiency. We may be able to call it a great achievement. But creating such beautiful code on our own may not come easily at first. The trick is to keep coding and producing work. And eventually, we make it through the plateau.

Some of the most successful people in the world were failures for a large majority of their lives. Yet we consider them geniuses. We have this unfortunate knack of ignoring the hard times they went through, their failures, and just looking at their successes. We also see them as they are now. Rarely do we look

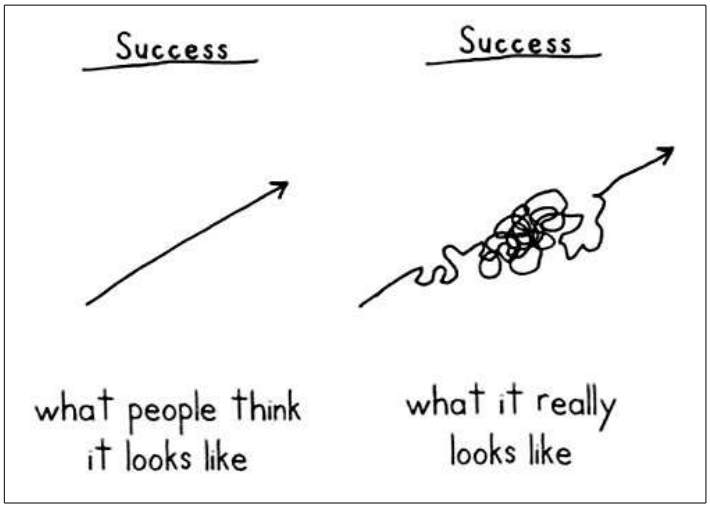


Figure 2: There are dozens of sources for this image, but none identify the original author.

Success is a journey, not a destination. – Ben Sweetland

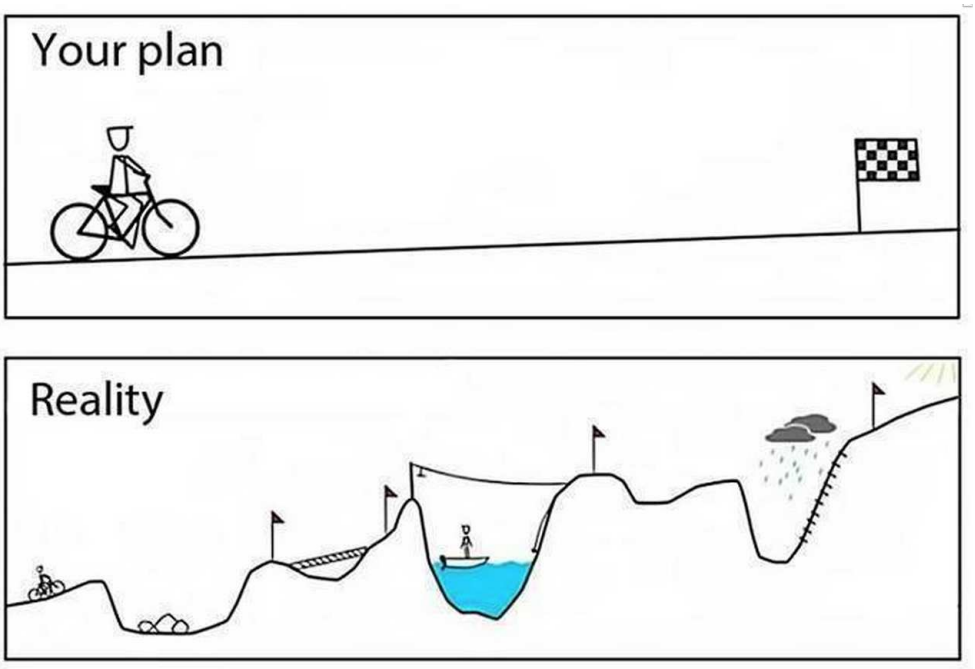


Figure 3: There are dozens of sources for this image, but none identify the original author.

at how long it took them to get to this point, or how much junk they put out before finally being successful. You would not believe how much utterly lousy code so-called experts created before finally creating something worth sharing and being proud of. You would not believe how much utterly lousy code most computer science professors wrote before finally producing something that they were proud of sharing with others without worrying if they would be laughed at.

It's not always obvious how much work it takes to become good at something. The perception is that success is achieved by following some straightforward path. The reality is that it's just not so simple. In our fast-paced world, we want immediate results and solutions right away. And when the solutions don't come quickly, we become discouraged. It is therefore hard to accept the idea of failure. But to be successful, we must understand that failure is a part of the process. It is inevitably on the path to success. The key is to learn from our mistakes (and the mistakes of others) in order to become better and to grow. And this helps lead to more successes than failures in the long run.

Surrounding garbage

To code is one thing. It takes skill, experience, and a whole lot of practice. But it actually takes a bit more. There is the added tedium of setting up our environment so that we can code efficiently. This means setting up an operating system, compilers, integrated development environments, the command line (terminal), and so on. Often, no one wants to help us set this stuff up because it's usually frustrating and takes time. There is a lot of surrounding stuff to learn to do in order to just get ready to begin to code. So we need to learn other things first in order to be able to even begin coding.

Spectacular achievement is always preceded by unspectacular preparation. – Robert H. Schuller

A lot of coding skill is about developing a knack for asking the right questions on Google. It's about knowing which results to filter out and which to take a closer look at. It's about knowing which code that we find by clicking around is best to use (and, of course, give credit to – we obviously don't want to plagiarize). We must become good at discovering patterns.

Feeling stupid

When it comes to coding, we should also get used to feeling lost and stupid. In fact, the anxiety of feeling lost and stupid is not something that we learn to conquer. It's something that we learn to live with and manage. The most common state for a programmer is a sense of inadequacy. There is a limitless amount of stuff to learn. We need to *constantly* learn new tools, new techniques, new principles, new hardware, and new software. We better develop a habit of liking to learn. In the end, it helps to be mentally prepared for feeling stupid.

Get set...

Learning computer science

Usually at first, our coding skills are pretty low. We take on a lot of projects that are fairly easy and designed to get us comfortable with coding and solving problems. As our skills grow, we begin to feel good. If the problems remain simple, however, we inevitably become bored. We aren't really getting any better by continuing to code simple stuff. It's like playing racquetball. If we limit our opponents to those that we can always beat, we won't become better at playing anytime soon. But as the challenges increase, we often begin to feel anxious. We feel inadequate and that our skills aren't good enough for more difficult problems. But if we keep solving more challenging problems and creating work, however, our skills will grow again. The trick is to manage our skills and challenges so that we continuously remain interested (see Figure 4). We want to tackle problems that are hard enough to test and grow our skills, but not easy enough to make us bored.

Kate Ray (technical co-founder of scroll kit, a visual web page creation tool; she now works for WordPress) has an insightful way of visualizing the learning process. She breaks it down into several steps:

1. Follow a tutorial, even if you don't understand everything that you're doing;
2. Rebuild the thing that you just made without the tutorial (at least, without using it too much);
3. Try to build something simple that you want to build that's related to the thing that you just built;
4. Find a new tutorial related to your new thing and use it to build this new thing;
5. Rebuild the new thing yourself without the tutorial;
6. Start a new project; and
7. Repeat 1 to 6 over and over again.

These steps seem to infer that we're doing this on our own. To a large degree, this is true. There is only so much of coding that we can learn from others. It is largely a skill that is grown by locking ourselves up in our rooms and creating more work. And we need to do this a lot on our own if we really want to *accelerate* the learning curve. It can be quite frustrating. But we need to learn to grind through the frustration. It helps to like to tinker and break things. It helps to be OK with not understanding everything, but to have the desire to understand everything. It helps to **aspire to intelligence rather than belittling it**. It helps to be OK with not always seeing our progress.

Unix for stability. Macs for productivity. Windows for solitaire.

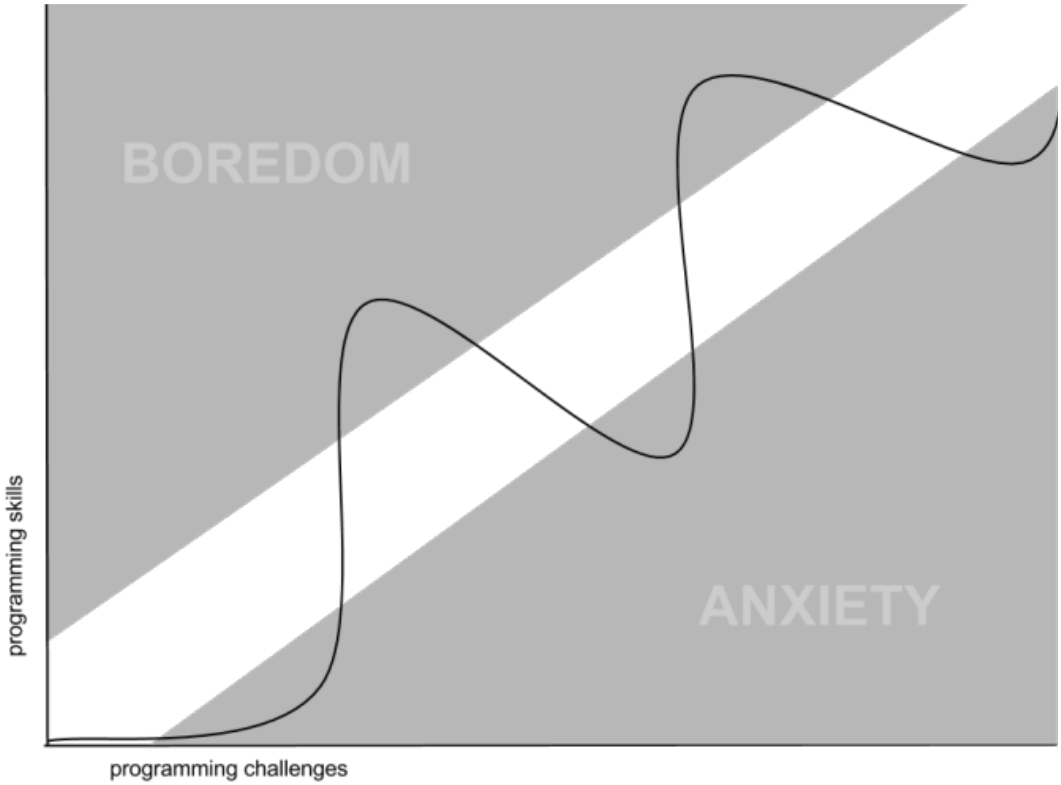


Figure 4: From an article by Kate Ray in TechCrunch.

And here's an observation: most people do not know how to learn, lack basic organizational skills, have absolutely no clue how to manage their time efficiently, and are grossly inefficient at processing and storing information. Truly, we all share some or all of these characteristics at one point or another in our lives, but very few people actually figure out how to regularly do these seemingly common things well. To develop these skills requires practice, often on our own. It's not particularly glamorous.

There are some things that we should always strive for. For example, we should try to learn about everything. Yes, this sounds a bit broad. More specifically, we should try to foster a desire to understand how things work and not just how to use them. We should try to be informed, both about things that we are interested in (like coding, maybe) and things that we know help to increase our awareness of the world and our usefulness in and contribution to society (like politics, maybe). Certainly this takes an immense amount of work on our part; but in the end, it is absolutely worth it. To help with this, the following is a little bag of tricks that I have developed over the years. It helps me solve problems, particularly those that relate to computing:

> Most people can't think, most of the remainder won't think, the small fraction who do think mostly can't do it very well. The extremely tiny fraction who think regularly, accurately, creatively, and without self-delusion – in the long run, these are the only people who count. – Robert Heinlein

Go!

A little bag of tricks

0. *What is the problem*? If you don't have a clear understanding of the problem, you will never be able to solve it. Period.
1. *Simplicity is, well, simple*. Give yourself a simpler problem. Try to find the trivial or base case. It greatly reduces the difficulty.
2. *Doodling is fun*! Draw pictures and diagrams to help you solve the problem. Few can think in math and formulas. Most of us are visual.
3. *Split it up*. Solving parts of the problem and then putting it all together often helps if that's possible. This is exactly what we do when we design complex algorithms.
4. *Work from both ends*. If you know what the outcome or solution is supposed to look like, then use that to help solve the problem. It could reveal ideas and shortcuts.
5. *Similarity, my dear Watson*. Many problems are similar in nature. Try to see if a problem is the same thing as another you've previously solved or know something about. There's nothing wrong with reusing old material if it is relevant.
6. *Does it make sense*? Often, students accept a first hack at an answer, forget to ask themselves this important question, and do not notice how nonsensical their answer might be. Ask yourself if an answer makes sense before you commit to it. This will avoid turning in assignments that contain unedited (and often unread) cut-and-pasted material from the Web with phrases like, "Click here for a detailed explanation." And yes, this has actually happened. This is probably the most ignored trick and the most annoying to a prof who is grading your assignment.
7. *If time permits, optimize*. It's never a bad thing to eliminate redundancy and optimize your solution to the problem.
8. *Don't give up*! Seriously, don't give up. If you need help, discussing general ideas with fellow classmates is always a good thing. However, you should refrain from discussing things that are too specific (e.g., code). Furthermore, teachers should always be glad to help you **if you show them that you have reasonably thought about the problem first**.
9. *Confidence is crucial*. A lack of it can actually be quite deadly.

Thanks to Matthew Michalewicz (a researcher in puzzle-based learning), here are a few rules that can help us solve problems:

1. Be sure you that understand the problem and all the basic terms and expressions used to define it;
2. Don't rely on your intuition too much. Solid calculations are far more reliable; and
3. Solid calculations and reasoning are more meaningful when you build a model of the problem by defining its variables, constraints, and objectives.

Epilogue

You may be wondering about what makes people good at coding or, more broadly, at computer science. Here are the kinds of characteristics that make *good* computer scientists:

1. *Curiosity*. Being inquisitive about the world helps us to understand and discover problems. Curiosity is helped along if we have broad and varied interests, a desire to want to know how the world works, and a desire to constantly want to learn new things.
2. *Creativity*. Thinking *sideways* (a term that means to think "outside the box") helps us to create new problem solving approaches and useful solutions. Our creativity grows as we get better at thinking critically using careful evaluation and judgment. It is helped along if we learn to use past experiences to solve current problems, to be socially competent in order to argue for our ideas, to work well in teams so that resources and ideas are shared, and to be good at (and, more importantly, to love) solving problems.

The world doesn't need more people with good grades. The world needs people who see the really tough problems as puzzles and have the tenacity and creative capacity to solve them. – Gever Tulley

3. *Focus*. Having perseverance and tenacity helps us work through long-lasting problems and allows us to deal with failure more easily. Focus is easier if we become good at recognizing patterns and at developing efficient solutions.
4. *Attention to detail*. In a way, this is related to focus and helps us to maintain rigor and find errors in our logic. We should assume that we aren't perfect (which we aren't), and that our solutions won't be perfect.
5. *Communication*. If life is about discovering solutions to hard problems, then we will undoubtedly have to convince others that our solutions work. We must therefore become good at communicating and defending our ideas both verbally and in writing.

So the above characterize *good* computer scientists. Well, here are the kinds of characteristics that make *great* computer scientists:

0. The kinds of characteristics that make good computer scientists.
1. *Organization*. Be organized and maintain detailed records.
2. *Time management*. Know how to manage your time efficiently.
3. *Efficiency*. Strive for efficiency when developing solutions.
4. *Improvement*. Desire to constantly improve – to become better at solving problems.
5. *The general case*. Design solutions that work for all cases instead of just one specific case. Aim for robust solutions instead of throwaway prototypes.
6. *Skepticism*. Question everything. It helps to develop and refine curiosity and creativity.
7. *Honesty*. Be honest with yourself and with others.
8. *Open mindedness*. You never know when someone else's ideas will trigger something that sparks a new (potentially better) idea.
9. *Continuously learn*. Understand that computer science is a lifelong learning process. There is no final level that denotes mastery of the discipline.
10. *Produce*. Understand that computer science is a production-oriented discipline. Trying hard is meaningless without eventually producing something that works. That takes a lot of repetition and practice.
11. *Confidence*. Although we constantly shift on the "novice-to-expert" scale, having confidence in our abilities wherever we sit on that scale goes a long way.

A final statement that is attributed to Larry F. Hodges (a fellow academic) and slightly paraphrased: a good thing to remember is that education in computer science is as much about learning how to think critically about issues and how to solve problems as it is about how to create and use technology. The technology is continually changing, but the problem-solving skills learned can serve a person throughout their life.

Bucket of quotes

I've heard a lot of quotes over the years. Here are some of my favorites:

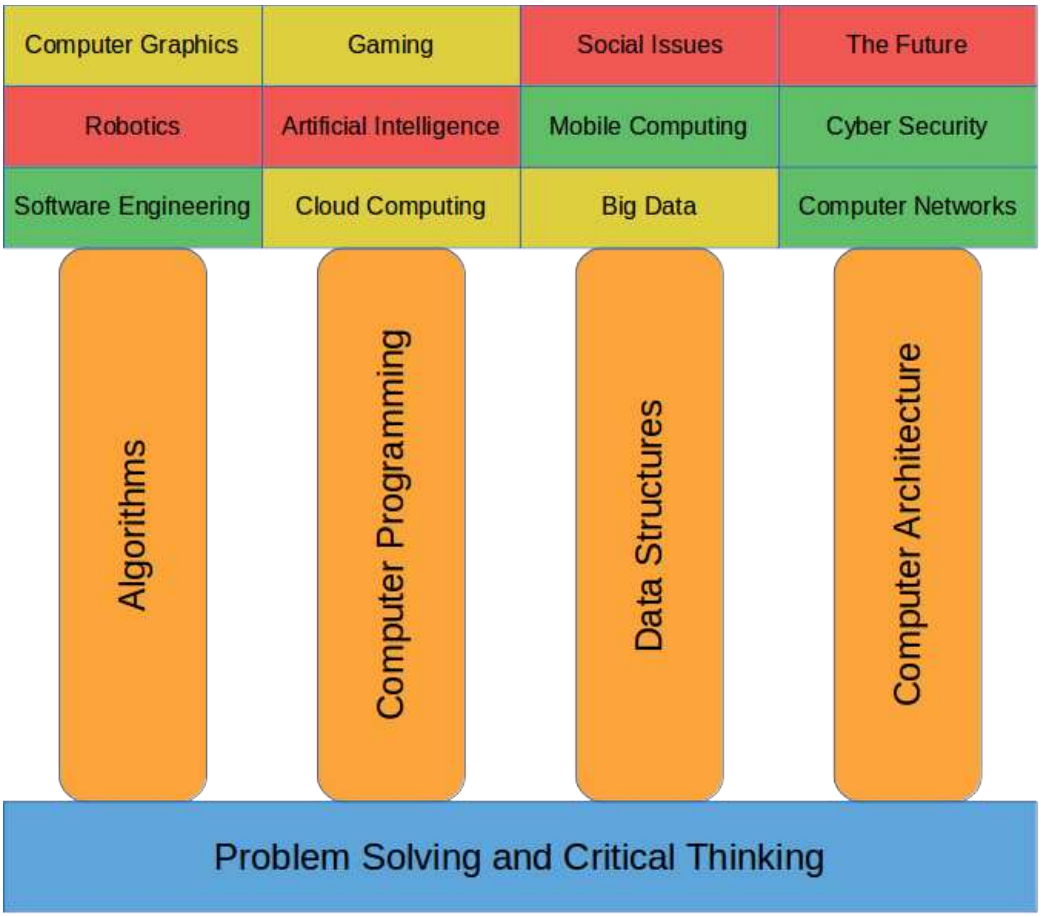
- Despite the cost of living, have you noticed how it remains so popular?
- Nothing is foolproof to a sufficiently talented fool.
- Light travels faster than sound. This is why some people appear bright until you hear them speak.
- Everyone has a photographic memory. Some just don't have film.

- Everyone makes mistakes. The trick is to make mistakes when nobody is looking.
- If you can't convince them, confuse them.
- Support bacteria – they're the only culture that some people have.
- It may be that your sole purpose in life is simply to serve as a bad example.
- Growing old is mandatory, growing up is optional.
- Make it idiot-proof, and someone will make a better idiot.
- Those who cannot change their minds cannot change anything. – George Bernard Shaw
- Your reputation is made by others. Your character is made by you.
- Tell the truth, there's less to remember.
- If you thought before that science was certain – well, that is just an error on your part. – Richard Feynman
- Don't go around saying the world owes you a living. The world owes you nothing. It was here first. – Mark Twain
- Success is just like being pregnant. Everybody congratulates you, but nobody knows how many times you were screwed.
- Life is like a jar of jalapeños. What you do today might burn your ass tomorrow.
- A successful life is one that is lived through understanding and pursuing one's own path, not chasing after the dreams of others. – Chin Ning Chu
- The difference between genius and stupidity is that genius has its limits. – Albert Einstein
- I have approximate answers and possible beliefs and different degrees of certainty about different things, but I'm not absolutely sure about anything. – Richard Feynman
- I would rather have questions that can't be answered than answers that can't be questioned. – Richard Feynman (attributed)
- Confidence is silent. Insecurities are loud.
- There are 10 types of people: those who know binary, those who don't, and those who weren't expecting it in base 3.
- If it is important to you, you'll find a way. If not, you'll find an excuse.
- I never lose. Either I win or I learn.
- If we don't believe in freedom of expression for people we despise, we don't believe in it at all. – Noam Chomsky
- True ignorance is not the absence of knowledge, but the refusal to acquire it. – Karl Popper
- If serving is below you, leadership is beyond you.
- You want to know the difference between a master and a beginner? The master has failed more times than the beginner has ever tried.
- You are not entitled to your opinion. You are entitled to your informed opinion. No one is entitled to be ignorant. – Harlan Ellison
- Stop Global Whining. – Jean Gourd

Our approach to computer science
The curriculum is approached from the bottom-up. That is, we first lay a **foundation** on which the **pillars** of computer science are *iteratively* built. We build them concurrently, a little at a time, growing them together until they stand tall. From there, we add top **beams** that provide further context by revealing more focused areas that represent the applications of computer science.

The foundation provides a solid platform of **problem solving** and **critical thinking** skills upon which to build the pillars. The curriculum implements a puzzle-based learning approach to assist in developing higher-order thinking skills (like problem solving and critical thinking) by incorporating specific puzzles (with relevant discussions) at regular intervals. The approach attempts to motivate students to think about how they frame and solve problems that they have never seen before. Strategies to problem solving are then applied more generally to problems in the domain of computer science.

The four pillars of computer science (Algorithms, Computer Programming, Data Structures, and Computer Architecture) are then iteratively built (i.e., a little at a time). They grow as the curriculum progresses, although perhaps not all at exactly the same rate; for example, computer programming may initially grow a bit faster than data structures. By iteratively building the pillars, we are able to quickly



The Living with Cyber curriculum.

cover the breadth of computer science. Although it is initially at a beginner's level, by this point you should have a good idea of what computer science is already. As they continue to be built, you will increase your knowledge and application of the pillars and how they combine to form the structure of computer science.

The first pillar is **Algorithms**. It forms the basis for representing solutions to problems. Algorithms allow us to formalize solutions to problems and represent them utilizing various formal methods. A large part of the curriculum focuses on the development of good algorithms as solutions to interesting problems. This allows us to refine our problem solving and critical thinking skills and focus them on the domain of computer science.

The second pillar is **Computer Programming**. We use computer programming to translate algorithms to a language the computer can understand. In the computer science curriculum, it is first experienced using the Scratch programming language, a puzzle-like way of creating computer programs. Scratch is a great beginner's programming language because of how it intuitively simplifies dealing with programming language syntax. From there, the curriculum quickly switches to Python, which provides a great way to get through the tedium of beginning computer programming without requiring a lot of background knowledge about various programming paradigms.

Data Structures makes up the third pillar. When writing programs, we usually make use of data structures to store and manipulate data that is required for our algorithms to work properly. The curriculum covers various data structures often used in computer programs. In addition, it focuses on many types of problems often found in computer science, their common solutions (as algorithms), and the various data structures that are often used in those algorithms.

The final pillar is **Computer Architecture**. Computers have a very well-defined architecture: physical, tangible hardware. The software (e.g., operating systems, applications, computer programs, etc) work intimately with the hardware; and this is what makes the whole thing work. The curriculum makes use of a unique hardware platform that you will use to work on hands-on projects throughout the curriculum. Currently, the platform is made up of a Raspberry Pi computer. The Raspberry Pi is a credit card-sized single-board computer that was developed for curricula that teach basic computer science in schools. Although it is a full computer, it is a bit slower than typical desktops and even laptops. The platform used in the curriculum also includes a touchscreen, a keyboard and mouse, USB-powered speakers, and other components to assist in designing external circuits.

The **beams** that sit on top of the pillars relate computer science to a variety of focused, derived topics. They primarily exemplify the application of computer science. Some examples include:

- Software engineering
- Cloud computing and big data
- High performance computing
- Computer networks
- Cyber security
- Mobile computing
- Robotics
- Artificial intelligence
- Computer graphics
- Gaming
- Social issues

- The future of computing

The end goals of the curriculum are straightforward. At the conclusion of the course, we expect students to:

1. Have knowledge of the breadth of computer science (and thus be able to answer the question, "What is computer science?");
2. Have the ability to solve a variety of problems and think critically through them;
3. Have the ability to generate algorithms as solutions to problems;
4. Have the ability to manipulate data through data structures;
5. Have the ability to translate algorithms to an object-oriented programming language; and
6. Have an understanding of the way computer hardware and computer software interact through a well-defined architecture.

What is Living *with* Cyber?

At its core, the freshman computer science curriculum is about problem solving and critical thinking. Therefore, the curriculum attempts to cultivate problem solving and critical thinking skills. Problem solving is one of those things that is useful in all of life, and arguably, is a necessary part of life. Obviously, it is the process of finding solutions to problems. And problems abound! Critical thinking allows us to objectively analyze and evaluate things that we've identified in order to form a judgment. Critical thinking helps us solve problems because problems have characteristics, variables, constraints, and so on that we must be able to identify, analyze, and evaluate, in order to derive a solution. And our solutions may not be unique. That is, better solutions may exist. So we must also be able to analyze our solutions and compare them to other potential solutions. These judgment calls allow us to find better solutions to problems.

Our solutions usually take the form of algorithms. Algorithms are like recipes with ingredients, a method of combining those ingredients, and some final dish. Who doesn't love apple pie? There are several recipes for apple pie. They aren't all the same, but they all make apple pie. Some have different ingredients (like butter vs. shortening vs. oil, Granny Smith apples vs. Macintosh apples, and so on) and some have a different way of combining those ingredients. So multiple recipes may exist for the same apple pie. Likewise, multiple algorithms will often exist as the solutions for a single problem. We must therefore be able to judge our solutions in order to determine first if they actually solve the problem, and second if they are the best solution to the problem. Evaluating algorithms (how efficient they are, how scalable they are, etc) is important. We may find better recipes that take fewer ingredients, or take less time to make, or make less of a mess in the kitchen, but all of which make apple pie. Ultimately, our solutions that take the form of algorithms are nothing more than recipes. They are steps that we can follow to solve the problem. Algorithms may have inputs (like ingredients), a process (like a method), and generate output (like an apple pie).

We often discover problems that are quite hard to solve manually. That is, we don't have enough energy or power (physical, mental, health, time, etc) to solve the problems ourselves. Therefore, we use machines like computers to solve them for us. But computers are dumb; they can only do what we tell them to do. We must therefore learn how to communicate with computers in order to be able to describe our solutions to a problem so that the computer can follow the algorithm that we've designed and solve the problem for us. Computers have much more processing power than we do. They can perform calculations and some types of actions far more quickly than we can as humans. For example, they can compute the square root of a large number much more quickly (but they can't paint our house...yet).

Living *with* Cyber is about solving problems by thinking critically through them, developing solutions to problems as algorithms, evaluating algorithms to ensure that they solve the problem efficiently, describing algorithms to a computer for execution using a programming language, and ultimately observing the result and asking, "Does it make sense?"

In order to be able to use computers to solve problems, we must then be able to communicate with computers. We must be able to specify an algorithm that the computer can understand and follow. But a computer doesn't understand an algorithm that we've written down on paper. A computer doesn't understand English. A computer actually only understands binary (1s and 0s). But we have a hard time understanding binary. We don't speak binary. So we've invented the idea of programming a computer using a high level programming language that's a little bit like English but that can be translated easily to binary.

Programming languages are different than spoken languages. English, for example, is a spoken language. It has rules, but those rules came about after the language had been spoken for a while. So there are exceptions (like i before e, except after c; or when you run a feisty heist on a weird beige foreign neighbor; or when caffeine-strung atheists are reinventing protein at their leisure; or when plebeians deign to either forfeit the language or seize it and reinvent it). It's also ambiguous. Take, for example, the phrase, "I made the robot fast." What does this mean? We may find four independent meanings. Perhaps it was built quickly. I made the robot fast. Or perhaps the robot was moving slowly and its designer wanted to make it move faster in a maze. I made the robot fast. Or perhaps the robot is named Fast. I made the robot, Fast. Or maybe the robot was temporarily convinced to stop eating nuts and bolts. I made the robot fast. We use contextual clues and intonation in spoken languages to provide hints at what something means if it is ambiguous. Programming languages aren't like this. And that's why it's a good thing that they are not ambiguous. We generate a programming language by first specifying its rules and then deriving the language from those rules. Programming languages are thus very precise. But because of this, we must learn a programming language's syntax (how to form sentences in the language) and also its constructs (the kinds of things that allow us to describe complex tasks in the language). Things like loop constructs that allow us to perform tasks over and over again, or decision making constructs, or constructs that allow us to combine a set of instructions in a logical group, or how it is that we can store and manipulate information. These are all important things that we must learn in order to be able to program in a high level computer programming language, so that we can describe our algorithms to the computer, so that they can be followed in order to solve problems.

Concretely, Living *with* Cyber is spread across the entire first (freshman) year in three courses: The Science of Computing I, The Science of Computing II, and The Science of Computing III. The first course attempts to answer the question, "What is computer science?" and begins the process of building the foundation and pillars of computer science. The other two courses continue to supplement the foundation and build up the pillars, while adding various top beams.

What is cyber?

Since the curriculum is called Living *with* Cyber, perhaps this is a good time to define *cyber*. Cyber is a shortened form of *cyber*space, which is a rather cloudy, fuzzy term that describes a space that is seemingly hard to actually define. But if we distill it down to a manageable level, we can define cyberspace as **the domain in which digital information moves**. That may still be hard to picture in our mind, however, so why don't we start with the Internet (which we seem to have more of a grasp of). The Internet is just a network of computers. And by network we mean that these computers are connected in some way. Sometimes these connections are actual wires (like Ethernet cables) or wireless waves that move through the air. Some of these machines look like the computers we use every day (e.g., laptops,

desktops, notebooks). Others look quite different, but they are still computers (although we may give them different names like routers, switches, intrusion detection systems, and so on).

If we picture the Internet as a large beach ball, then the World Wide Web would fit inside that beach ball and would be considerably smaller (like a softball). The World Wide Web is what we use when we browse web sites. Web browser like Edge (formerly Internet Explorer), Mozilla Firefox, Safari, and Google Chrome allow us to browse through the World Wide Web. But the Internet also contains other softball- and tennis ball-sized things like e-mail, FTP, SSH, telnet, remote desktop, IRC, and so on. All of these words and acronyms are simply other kinds of services and entities that exist in or are subsets of the Internet. We may cover some of these later on.

Cyberspace, then, is the size of a large room. It is much more than the Internet. We can describe what kinds of other things might be in the domain of cyberspace. Take, for example, a buoy in the ocean that monitors wave height. These buoys are useful, particularly when storms (like hurricanes) are in the area and are expected to hit land. By monitoring the height of waves as the storm approaches, we can try to predict the surge that the storm will produce (how much water the storm will blow on land). These buoys transmit their data (wave height) to some monitoring center. These buoys and monitoring centers are part of cyberspace!

Take, as another example, an herb garden in someone's backyard. It requires care (e.g., watering and weeding). But perhaps its caretaker likes to go on vacation sometimes. So how does the garden get watered? Sure, someone could be assigned the temporary responsibility to water it for its owner (or hope that it rains enough but not too much). But this garden is special. It has moisture sensors in the dirt at several places in the garden. These sensors monitor how much water is in the soil. They provide a numerical reading (value) to a small low-power computer. This computer then periodically polls the sensors and controls a solenoid (an electrically-controlled valve) that can disperse water in the garden. The computer is connected via wireless to the Internet and can therefore remotely monitor the garden! Individual moisture sensors can even be remotely polled, and the solenoid can be remotely opened or closed! These sensors, the solenoid, and the low-power computer are all part of cyberspace!

And so are cell phones and home security systems (that allow remotely unlocking a door, turning off a motion sensor, etc), and time lapse cameras on rooftops taking periodic shots of a new building going up, and satellites, and the Mars rover, and so on...

Cyberspace is everywhere and involves so many things. It is an integral part of our daily lives. It is involved in transportation, shipping, banking, controlling critical infrastructure, and so on. There is hardly a time that it doesn't somehow get involved in our lives. And this is predicted to become more involved in our lives in the future! So now we see why it is important that we look at and attempt to solve the problems that arise in this domain! Living *with* Cyber attempts to cultivate problem solving and critical thinking skills by taking a look at and investigating problems that are in the important domain of cyberspace.

What is computer science?

If you stopped someone on the street and asked him "What do computer scientists do?" you would probably get a response along the lines of "They work with computers."

Most people know that computer scientists work with computers. Many are less clear when it comes to knowing exactly what computer scientists do with those computers. Some people believe that computing involves learning to use spreadsheets, word processors, and databases. The truth is that

computer scientists are really no more likely to be proficient with word processors or spreadsheets than any other professionals. Some people think that computer scientists build or repair computers. While computer scientists do study the physical organization of computers and do help to design them, they are not the people to call when your disk drive won't work.

Many people know that computing has something to do with writing computer programs. In fact, it is a commonly held notion that the study of computer science mainly involves learning to write computer programs. While it is true that computer scientists must be proficient programmers, the field is much broader than just programming.

Programming is probably best thought of as a skill that computer scientists must master, just as a carpenter must master the use of a hammer. In the same way that carpentry is about building things and not about simply pounding nails, computer science is not about programming. Rather, it is about solving problems through the design and implementation of computing systems. Programming is only one small, rather mechanical, step in this process.

This curriculum is about the science of computing, which is also known as computer science or, more simply, computing. Computing is a broad topic that combines aspects of mathematics and engineering with ideas from fields as diverse as linguistics, art, management, and cognitive psychology. The people who work in this field are known as **computer scientists**. At the heart of the field is the concept of an **algorithm**, which is a detailed sequence of steps that describes how to solve a problem or accomplish some task.

Computer scientists are essentially problem solvers. Although they are capable of writing the programs that perform various applications such as word processing or data storage and retrieval, that is not their primary function. Instead, they study problems in order to discover faster, more efficient algorithms with which to solve them. Computer scientists also study and design the physical machines, called **computers**, that carry out the steps specified by algorithms. In addition, they design the programming languages people use to communicate algorithms to computers. In a very real sense, **computing** is the study of algorithms and the data structures upon which they operate.

Students who study computer science usually go on to take jobs that require the ability to program. Some people go into high technology application areas such as communications or aerospace. Others end up designing computer hardware or low-level “systems” software. Still others end up working in computer graphics, helping to develop software for medical imaging, Hollywood special effects, or game consoles. The possibilities are almost endless.

Our study of computer science will be guided by four questions, which, taken together, give a good overview of computing. These questions are:

1. How are computers used?
2. How does computer software work?
3. How does computer hardware work?
4. What are the limitations and potential of computing?

How are computers used?

Computers are integral to the fabric of modern life. Without computers there would be no cell phones, no ATMs, and no game consoles. There would be no widely accepted credit cards. Air travel would be much less common, much less safe, and far more expensive. In fact, without computers many aspects of modern life would be missing, and those that remained would be much, much more expensive.

Computers were originally conceived of as machines for automating the process of mathematical calculations. Given this fact, it is more than a little surprising that many of today's most popular computer applications focus on communication rather than computation. Aside from applications designed to help people communicate with one another more effectively, entertainment is probably the second most popular use of computing technology. Entertainment includes applications such as the production of Hollywood special effects and computer/console games.

Gaming is tremendously important to computing. The desire for greater and greater levels of realism pushes forward 3D computer graphics hardware, modeling software, artificial intelligence applications and many other aspects of the field. Gaming also brings many new people to computing. It is not uncommon for students to become interested in computer science due to a desire to create new and better games.

Another extremely useful, but somewhat more mundane, computing application is the spreadsheet. Before the use of spreadsheets became commonplace in the mid-1980's, using a computer to solve almost any task, such as averaging grades or balancing a check book, required the development of a unique computer program for that task. This led to great expense, since creating a computer program can be a difficult and time consuming task that requires the services of skilled professional programmers. With the advent of spreadsheets many simple applications no longer required that custom computer programs be written.

A **spreadsheet**, such as Microsoft Excel or LibreOffice Calc, is a program that allows a person to quickly model a collection of quantitative relationships without writing a computer program. Spreadsheets are common in the business world. They are used to tabulate data, perform long series of calculations, and to answer "what if" type questions that require mathematical modeling. If this sounds overly complex, don't be put off. While spreadsheets are one of the most powerful and flexible applications of computers, they are also one of the most intuitive.

Information storage and retrieval is another important application of computing that has existed since the earliest days of the field. After the federal government, large businesses were one of the first groups to adopt computers. They did so in the late 1950s in order to improve the efficiency of their accounting and billing operations. Without computers it would be impossible to conduct business in the way we do today. Just imagine the number of employees your bank would need if it had to process all checks by hand. The costs involved would make modern banking an unaffordable luxury for the vast majority of people. Computers were able to streamline financial operations because people quickly recognized that computers could store and rapidly retrieve vast amounts of data. In fact, the popular press of the 1950s often referred to computers as "electronic brains" because of their highly publicized speed and accuracy.

Often, data is stored in individual **files** of information. In addition to this, database management systems are useful for organizing large collections of related information. A **database management system** is a program that can be used to organize data in ways that make it easy for people to pose questions. For example, a well-organized database would make it easy for the dean to find answers to questions such as "How many seniors have a GPA of 3.5 or above?" or "Which classes did students attend at least 99% of the time last year?"

There are also a number of issues of interest to the general public that lie in the domain of computer science and are interesting to take a look at. Take, for example, security and privacy. **Security** generally refers to how well information is protected from unauthorized access, while **privacy** is concerned with

what information should be protected and from whom. Unauthorized access to your bank account is a security issue. Whether the government has a right to monitor electronic communications is a privacy issue.

Because computers are able to rapidly process, store, and retrieve vast quantities of information, they have always posed a serious potential threat to individual privacy. Before computers were commonplace, personal information such as an individual's name address, phone number, income level, and spending habits could be collected, but the expense involved in doing so by hand was generally prohibitive. Computers make the compilation of detailed personal profiles, from what kind of toothpaste you use to your political orientation, both practical and inexpensive.

As computers become more and more interwoven into the fabric of society, issues of security and privacy take on greater urgency. Do governments have the right to monitor private electronic communications? What constitutes a "private" communication? Do employers have the right to monitor their employees' email? Do teachers and administrators have the right to monitor email sent by students?

How does computer software work?

Now that we have talked about some of the many things computers are used for, the question that naturally arises is "How are computers able to accomplish all of these different tasks?" The first thing to understand is that computer systems are composed of two major components: hardware and software. **Hardware** consists of the actual electronic components that make up a computer. Unlike hardware, computer software is completely abstract. It has no physical existence. You cannot see, hear, touch, smell, or taste software. Although it has no physical existence, software is real – as real as the color red, or the number five, or the concept of beauty. One way to think of software is as "codified thought."

More formally, **software** consists of computer programs and the data on which they act. **Computer programs** are algorithms expressed in a form that is executable by computers. As mentioned earlier, an **algorithm** is a detailed sequence of steps that specifies how to solve a problem or accomplish some task. So, computer programs are just sequences of instructions that a computer can follow to solve a problem or accomplish a task.

It is important to note that the terms "program" and "algorithm" are not synonymous. A description of how to alphabetically order a list of words is an algorithm. This ordering procedure is not a computer program, since it is written in English, rather than being expressed in a form that can be recognized by a computer.

Computer programs must be written in programming languages, such as Scratch, Python, Java, C++, and so on. **Programming languages** are formal systems for precisely specifying algorithms and their associated data structures in a form that can be recognized by a computer. By "formal," we mean that these languages are based on sets of rules that dictate how instructions must be formed and interpreted. Programming languages are designed in such a way as to eliminate the ambiguity for which natural languages, such as English, are notorious.

Computer programs operate on data. **Data** are the symbols, usually characters or groups of characters, used to represent information that has meaning to people. The words and pictures making up this lesson are data. The meaning they convey to you as you read them is information. One of the fascinating aspects of computers is that they have the ability to manipulate symbols without needing to understand

the meaning of those symbols. A word processing program allows us to edit this text, even though it cannot read English and has no notion of the meaning of these words.

Computer programs read input data, manipulate that data in some way, and produce output. A program of any complexity will need ways of organizing, or structuring, that data. Hence, an understanding of data structures is critical to understanding computer software. A **data structure** is a collection of data together with a group of operations that manipulate that data in certain predefined ways. For example, the **queue** is a standard data structure that models a waiting line. It supports two basic operations enqueue and dequeue. "Enqueue" adds an item to the end of the waiting line. "Dequeue" removes an item from the front of the waiting line.

We can also examine the major types, or paradigms, of programming languages. A **paradigm** is a way of thinking about a problem or modeling a problem solution. There are at least three identifiable programming paradigms: the imperative paradigm, the functional paradigm, and the logical paradigm. Some computer scientists view object-oriented programming as a fourth paradigm. Others prefer to view objects as an extension to the imperative, functional, and logical paradigms. The vast majority of programs are written in imperative languages. Python, C++, Java, Fortran, C, Pascal, Ada, Modula-2, and COBOL are all imperative languages. The **imperative paradigm** derives its name from the fact that statements in imperative languages take the form of commands. In English, imperative sentences, such as "eat your vegetables" or "take out the trash" are commands where the subject is understood to be "you" – "You take out the trash." Imperative programming languages are similar in that each statement is a command instructing the computer to perform some action.

The **functional paradigm** is patterned after the mathematical concept of a function. A **function** defines a mapping from inputs (i.e., the domain) to outputs (i.e., the range) in which a particular input sequence always maps to the same output. Addition is an example of a function. Simple addition takes a sequence of two numbers as input and produces a single number as output (e.g., 5 + 4 = 9). Notice that since addition is a function, the same input sequence always produces the same output. This means, for example, that 5 + 4 must always equal 9 and never anything else. While a particular input sequence must always generate the same output, it is often true that a function will map many different input sequences to the same output. So, while 5 + 4 must always equal 9, so can 6 + 3 and 7 + 2 and 993 + -984. Another key characteristic of functions is that they always return one, and only one, result.

In the functional paradigm, statements are functions to be evaluated. This is different from the imperative paradigm in which statements are commands to be executed. As a result, functional programs tend to take on the form of nested expressions, rather than sequences of commands. Another difference between imperative and functional programs is that imperative languages generally store the results of computations in declared variables. Functional languages avoid the use of declared variables. Instead, values are computed as needed. The following example will help to illustrate these differences; here is a code fragment written in the imperative style:

```
read(tempc);
tempf = (1.8 * tempc) + 32;
write(tempf);
```

Now here is a code fragment written in the functional style:

```
(write(add (multiply 1.8 (read)) 32))
```

This example illustrates two implementations of an algorithm for converting temperature readings from Celsius to Fahrenheit. Both code fragments do the same thing: read a temperature; multiply that temperature by 1.8; add 32 to the result; and then display the final answer. Don't worry if some of the details of this example elude you. At this point all you need to recognize is that the various paradigms can produce quite different programs, even if those programs are based on the same underlying algorithm.

The **object-oriented approach** adds the concepts of objects and messages to the above paradigms. Essentially, programs and the data on which they act are viewed as objects. In order to perform a task an object must receive a message indicating that work needs to be done. The object-oriented approach is well suited for implementing modern "point and click" program interfaces. These interfaces consist of a number of graphical symbols, called **icons,** that are displayed on the screen. Whenever a user clicks the mouse pointer on one of these icons, such as a button or scrollbar, a message is sent to the corresponding program object, causing it to execute.

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. 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 long distance 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. **Software engineering** is the study of the design, construction, and maintenance of large software systems.

How does computer hardware work?
All general-purpose computers, at a minimum, consist of the following hardware components: a central processing unit, main memory, secondary storage, various input/output devices, and a data bus. A diagram showing the major hardware components is presented in Figure 1.

The **central processing unit**, or CPU, is the device that is responsible for actually executing the instructions that make up a program. For this reason, it has sometimes been referred to as the *brain* of the computer. **Main memory** is where the programs and data that are currently being used are located.

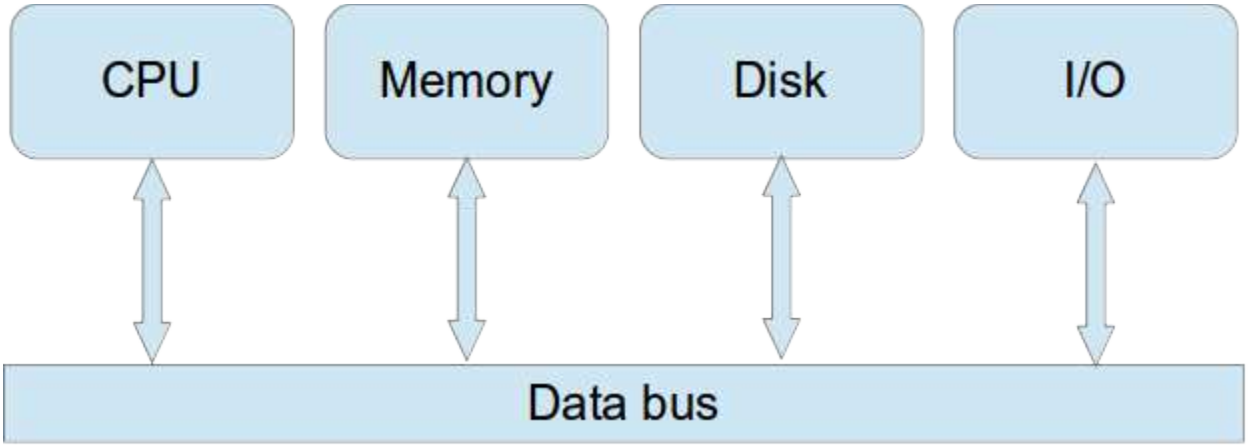


Figure 1: A block diagram of a computer

Main memory is often referred to as **RAM**, which stands for Random Access Memory. This acronym is derived from the fact that the CPU may access the contents of main memory in any order – there is no fixed or predefined sequence. In 2015, a new personal computer typically had between two to eight gigabytes of main memory, meaning that they could store somewhere between two to eight billion characters.

Secondary storage is used to hold programs and data that are likely to be needed sometime in the near future. Disk drives are the most common secondary storage devices. The capacity of secondary storage devices purchased in 2015 ranged from about 500 gigabytes to one terabyte, meaning that they could store somewhere between 500 billion to one trillion characters. The storage capacity of memory devices, both main memory and secondary storage, tend to increase rapidly over time. Historically, they have doubled approximately once every 18 months. This observation, known as **Moore's law**, has remained true since the introduction of computers more than half a century ago. Moore's law also appears to apply to the speed at which CPUs process instructions. Personal computers purchased in 2015 operated at speeds of approximately 3.5 billion cycles per second (3.5 Gigahertz) – meaning they could execute almost 3.5 billion individual instructions per second. It is this blinding speed that allows computers to accomplish the amazing feats they are capable of.

While the actual sizes of main memory and secondary storage continue to rapidly increase, their relative characteristics have remained fixed for at least a quarter century. Historically, secondary storage devices have tended to hold about 100 times as much information as main memory. In general, main memory is fast, expensive, and of limited size when compared to secondary storage. Conversely, secondary storage is slow, cheap, and large compared to main memory. The primary reason for these differences is that main memory consists of electronic components that have no moving parts. Secondary storage generally involves electromechanical devices, such as a spinning disk, on which information may be read or retrieved using magnetic or optical (laser) technology. Because these devices contain moving parts, they tend to be many times slower than main memory. However, recent advances have been made that address this by removing the moving parts from secondary storage (e.g., solid state drives). Another difference between main memory and secondary storage is that secondary storage is persistent, in the sense that it does not require continuous electrical power to maintain its data. The main memory of most computers is, on the other hand, volatile. It is erased whenever power is shut off.

The **data bus** is the component of a computer that connects the other components of the computer together so that they may communicate and share data. For example, the instructions that make up a computer program are usually stored in main memory while the program is running. However, the actual computations take place in the CPU. Before an instruction can be executed, it must first be copied from the main memory into the CPU. This copying operation takes place over the data bus.

In order for a computer to do any kind of useful work, it must have ways of communicating with the outside world. **Input/output devices** allow computers to interact with people and other machines. I/O devices range from the mundane (e.g., keyboard, mouse, and display) to the exotic (e.g., virtual reality glasses and data gloves). Some devices, such as keyboards, are strictly for input only. Other devices, such as display screens, are output only. Still other devices, such as modems, can handle both input and output. The general trend in I/O devices is towards higher throughput rates. Newer I/O devices can transmit and/or receive greater quantities of data in shorter periods of time. This trend is related to the desire to incorporate very high-resolution graphics, sound, music, and video into modern software products – all of which require large amounts of memory and rapid I/O.

Although modern computer hardware is quite impressive, it is easy to overestimate the capabilities of these machines. Computers can directly perform only a small number of very primitive operations and, in general, their CPUs are sequential devices – able to perform only one instruction at a time. These limitations are not apparent to the average computer user because computer scientists have built up a number of layers of software to insulate the user from the physical machine. Software can be used to make computers appear much more capable and friendly than they actually are because of the tremendous speeds at which they operate.

What are the limitations and potential of computers?
When trying to determine the limits of computing, two separate but related questions are generally of interest: “What problems can be solved in practice?” and “What problems, by their very nature, can never be solved?” The answer to the first question changes over time as computers become more powerful. The answer to the second question does not change.

Generally, the amount of time a program takes to run, together with the amount of space it requires to perform its computations, are the characteristics that are most important in determining whether the program is “practical.” If a program requires 48 hours to forecast tomorrow's weather, it is not very practical. Likewise, a program will be of little use if it requires more memory than is available on the computer for which it was designed.

There are often many different ways to solve a specific problem. When there are multiple algorithms for solving a problem, and they all produce identical results, people generally prefer the algorithm that runs in the least time (or, sometimes, that uses the least space). Determining the *best* algorithm for solving a problem can, however, be a difficult issue, since the time and space requirements of different algorithms increase at different rates as the size of the problems they are solving increase. In other words, given two algorithms A and B, algorithm A may be faster than B on problems of size 10, but B may be faster than A on problems of size 100 or larger.

Computer scientists have developed methods for characterizing the efficiency of computer programs in terms of their time and space requirements, expressed as a function of the size of the problem being solved. What is most important about an algorithm is usually not how much time it takes solving a specific instance of a problem (e.g., ordering a particular list of names), but instead the manner in which compute time and memory requirements increase with increasing problem size (i.e., how quickly do the

algorithm's resource needs increase when presented with larger and larger input lists). The reason for this focus on *rates of increase* rather than absolute numbers is that computer programs are generally not used to solve the same exact instance of a problem over and over, but instead to solve many different instances of the same problem.

To address the question of "What problems can never be solved," computer scientists design and study abstract models of computers. No thought is given to designing practical machines; the models are purposefully kept as simple as possible. This is done in order to ease the process of analyzing what problems these models can and cannot solve. Even though computer scientists have identified some problems that they know cannot be solved by computers, in general they are not sure what the practical limits of computing are. Currently, there are many tasks at which computers far surpass humans and there are others at which humans excel. The problems humans find difficult (such as performing a long series of calculations) tend to have straightforward computational solutions, while problems which humans find trivial (such as using language and comprehending visual information) are still very difficult for computers.

What does the future of computing look like?
Obviously, no one knows what the future holds; but using history as a guide, educated guesses can be made. In thinking about the future of computing, we look at the near term (say within the next decade) and then further out (about three decades from now).

For the near term one can be relatively certain that computing speed and memory density will continue to increase. Confidence in this prediction comes from current laboratory work and the past track record of the computing industry at translating research results into mainstream computing products. It is also likely that computer networks will continue to increase in bandwidth for the same reasons.

To get some sense of where computing may be in thirty years, we again turn to Moore's law. **Moore's law** states that computing power and memory density double roughly every eighteen months. This "law" is just a future projection based on past performance – but as they say in investment circles "historical performance is no guarantee of future advancements." Some computer scientists do not believe that the rate of progress predicted by Moore's law can be maintained much longer. They point to the fundamental limits of silicon-based technology and the ever-increasing costs of developing the next generation of computer chips. Others are more optimistic. They point to ongoing research projects that seek to develop fundamentally different approaches to computing. These approaches, which have the potential to increase computing speeds by many orders of magnitude, include **optical computing**, which seeks to build computers that use photons (light) instead of electrons (electrical current); **biocomputing** in which strands of DNA are used to encode problems, and biological and chemical processes used to solve them; **nanocomputing** which aims to build molecular-level computing systems; and **quantum computing** which seeks to exploit the characteristics of quantum physics to perform computations.

Assuming the optimists are right and the trend embodied in Moore's law continues to hold, what will computers be like in thirty years? The numbers suggest that the average desktop computer will have the processing capacity of the human brain together with a memory capacity many times that of a human being. The result could be human-level intelligence on your desktop within your working lifetime. Take a moment to contemplate this. In thirty years you will probably just be reaching the height of your professional career, and you may be dealing, on a daily basis, with computers that rival humans in computational power. The implications are truly awe inspiring – and somewhat frightening.

How will society deal with such machines? Should they be afforded the same rights as individuals? Or will they be considered as mere possessions? Is it moral to construct such machines simply to do our bidding? Or will the practice of owning machines that can "think" be thought of as barbaric as the concept of slavery? Though these questions may seem far-fetched now, some people think there is a good chance that we will have to come to grips with these issues during our lifetime.

Of course, even if Moore's law does continue to hold and machines with the processing capacity of humans are developed, that does not mean computer scientists will have necessarily figured out how to effectively use these resources to solve the problems of artificially embedding intelligence into computers. In other words, although we may create machines that can do the same amount of raw computational work as the human brain, we may never figure out how to program them to be *intelligent*.

Homework

See the *What will the future of computing look like?* document.

In this lesson, you will learn about one of the most fundamental concepts in computer science: algorithms.

Definition: *An **algorithm** is a detailed sequence of steps that describes how to solve a problem or accomplish some task.*

First, consider the process of identifying problems and their possible solutions, and breaking those solutions down into a *sequence of steps*. One way to do this is via simple to-do lists. Consider the problem of getting to class in the morning. Assume that you have just woken up (that is, after you hit the snooze button a good dozen or so times). What steps did you take to prepare and get to class? A possible list of steps is:

```
get out of bed
brush teeth
take shower
get dressed
eat breakfast
grab book-bag
walk/bike/drive to class
```

This to-do list is actually a useful way to tell someone who didn't know what to do in the morning to prepare to get to class, what to do.

Another example of a problem is that of baking an apple pie. What would a to-do list that would show someone who didn't know how to bake an apple pie look like? Could there be multiple ways of baking apple pie? If so, how can we determine which one is the *best*? Truly, this is subjective; however, the idea of picking a best solution is one that is required in computer science.

A possible solution to baking an apple pie as a to-do list may look like this:

```
make the dough
make the apple goop
combine the two
bake the dough with the goop
eat hot apple pie (with vanilla ice cream, of course)
```

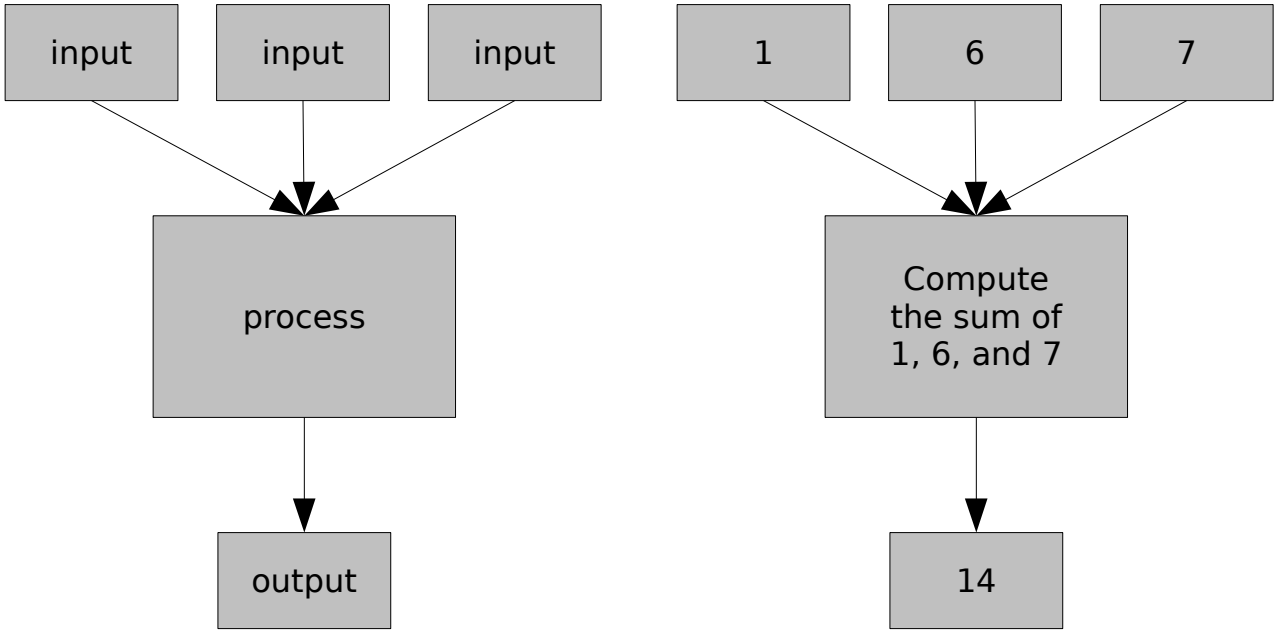
What does an algorithm that describes how to bake apple pie in the *best* manner mean? Is it the algorithm that takes the shortest time to produce hot apple pie? Is it the algorithm that uses the fewest ingredients? Is it the algorithm that makes less of a mess in the kitchen and therefore requires less cleanup? These questions are designed to illustrate that the notion of *best* algorithm is many times quite subjective and does not always have a definite (or even a single) answer. However, the comparison of algorithms numerically to show the performance of one against another is quite important in computer science. This will help in determining if one is better than another, with the end goal being to identify a *best* algorithm.

Does a wrong algorithm for producing apple pie exist? What solutions or algorithms will not work? What happens if a person uses blueberries instead of apples? Is the original problem of baking an apple pie solved then?

Problem statements: wrong vs. right algorithms
It is important to define standard ways of representing problems so that anyone who reads about a problem can understand what it actually is.

Definition: *A **problem statement** is a formal way of defining a problem that contains a description of the conditions at the start of the problem solving process (also known as inputs), and a description of the valid solutions (also referred to as outputs).*

For example, if a problem was to add three numbers and produce the sum, what would the inputs be? The answer, of course, is the three numbers! What would the output be? Clearly, the sum of those three numbers. If a problem was to determine the amount of income tax owed this year, what would the inputs be? Your income. And the output? The amount of tax owed this year. The following figure illustrates the sum example (generally on the left, and specific to the sum of three numbers on the right):



Consider the *producing apple pie* problem. What would the possible inputs for the algorithm be? What would its valid (or correct) output(s) be? One way of telling if an algorithm is correct is whether or not it produces the valid output defined in the problem statement. If an algorithm produces blueberry pie when the output statement stated that it was required to produce apple pie, then that algorithm is a wrong solution. On the other hand, if the output is an apple pie (even if the algorithm instructs you to throw away your ingredients and then buy an apple pie from the grocery store), then that algorithm is technically correct (even though it may not be very *efficient*).

Step breakdown and control flow
Consider the *get to class* algorithm. It consisted of steps such as *brush teeth*, *get dressed,* and *eat breakfast*. Some of these steps can actually be defined as problems of their own, which themselves require different algorithms in order to be successfully completed. What this means is that some of

these steps can be broken down into their own algorithms with a separate to-do list. We call these sub-steps. A possible algorithm for the problem of *getting dressed* is:

```
put on underwear
put on shorts
put on shirt
put on socks
put on shoes
```

This illustrates that we can *zoom in* to any step in an algorithm and break it down into a series of smaller steps. Note that it is possible to arrive at a trivial step that does not require any smaller steps or that does not have a simpler way to describe it.

So what should be done to our original *get to class* algorithm? Should *getting dressed* be replaced with the five new sub-steps identified above? Is the overall solution more correct or less correct if we do that? In the end, both solutions work and neither is specifically better than the other. It all depends on the desired level of detail. If the person for whom the algorithm was intended for knew how to get dressed, then a separate algorithm to explain that would not be needed. However, if the person didn't know how to get dressed, then these sub-steps would be very important.

Let's go back to the *producing apple pie* problem. What do you think would happen if the order of the steps was changed in the proposed solution? What if the raw ingredients used to make the dough and goop were baked *before* putting them together? In this case, there would be a very messy oven instead of an apple pie. What if the apple goop was made before making the dough? In this case, apple pie would still be the delicious result. There are times when the order in which the steps of an algorithm are carried out matters a lot and other cases when it doesn't matter as much.

Consider the *get to class* algorithm again. What would happen if the order of the steps was changed? Imagine getting dressed before taking a shower. Imagine walking/biking/driving to class before getting dressed. Let's not. Could a shower be taken before brushing teeth? Could breakfast be eaten before brushing teeth?

Definition: ***Control flow*** *is the order in which the instructions in an algorithm are evaluated or executed.*

In the examples considered so far, the control flow of algorithms has been the order in which the steps were listed. Each step is evaluated or performed until it is complete, and then we move on to the next step. Completing all the steps completes the to-do list, thereby completing the algorithm. Numbering the items listed in a to-do lists is an easy way of detailing the order in which the steps should be completed. A possible numbered solution to the *get to class* algorithm is:

```
1: get out of bed
2: brush teeth
3: take shower
4: get dressed
5: eat breakfast
6: grab backpack
7: walk/bike/drive to class
```

We say that *flow of control* begins at step 1, flows to step 2, then to step 3, and so on. This means that step 1 is completed first before attempting step 2. Step 5 should not be done before step 4 is completed.

What about the *get to class* algorithm in which separate sub-steps for *get dressed* were specified (step 4 in the algorithm above)? To combine the two algorithms, numbers can be assigned to the steps in the *get dressed* algorithm to show the order in which they will be executed *in relation* to the *get to class* algorithm – which will now look like this:

```
 1: get out of bed
 2: brush teeth
 3: take shower
 4: get dressed
10: eat breakfast
11: grab backpack
12: walk/bike/drive to class
```

The *get dressed* algorithm will now look like this:

```
5: put on underwear
6: put on shorts
7: put on shirt
8: put on socks
9: put on shoes
```

Notice how steps 1 through 4 of the *go to class* algorithm are the same and done in the same order. However, we *shift* the flow of control to the *get dressed* algorithm which used to be step 5 in the original. We have to execute the *get dressed* step (meaning its sub-steps) completely before proceeding to the *eat breakfast* step. However, the *get dressed* step consists of five sub-steps, and so we number those steps before continuing to the *eat breakfast* step. This means that the first sub-step in the get dressed algorithm is step 5. We have to complete the *get dressed* algorithm before returning to the *get to class* algorithm. Thus, the *eat breakfast* step becomes step 10 since the last step of the *get dressed* algorithm is step 9. The entire algorithm ends when we execute the last step of the *get to class* algorithm, which is step 12.

Flowcharts

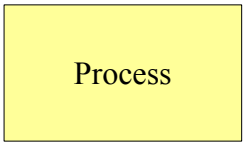
Up to this point, to-do lists have been used to represent algorithms. There are other ways of representing algorithms, however, and one of them is by using flowcharts.

Definition: *A **flowchart** is a type of diagram that represents an algorithm, listing steps with various blocks and flow with arrows.*

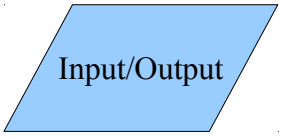
Flowcharts are made up of different types of *blocks*, each of a different shape. The shapes correspond to different kinds of statements or types of item. Flowcharts use arrows to show the direction of execution of a given algorithm. An arrow leading from a block A to a block B means that A is executed before B.

To symbolize where the algorithm starts and ends, we have a special kind of block called the **terminal block**. Terminal blocks are oval in shape and identify where an algorithm starts and where it ends. The start block will always have one arrow leading out of it to the next block to be executed, and the stop block will always have at least one arrow leading into it from some previous block. Here is what a terminal block looks like:

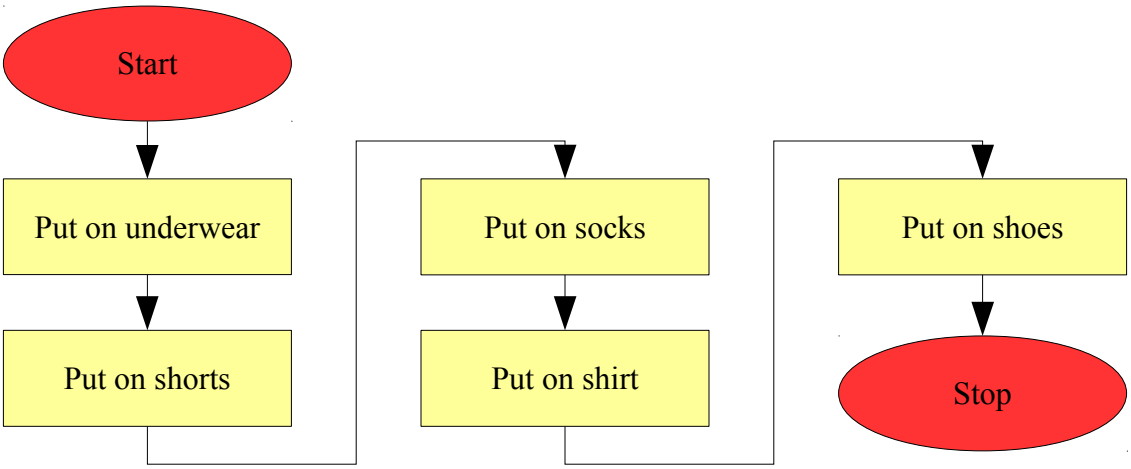
A **process block** represents statements in which some action is performed. It is shaped like a rectangle. Typically, this block will have one arrow leading into it from a block that was executed before. It will also have an arrow leading out of it to the block that should be executed next. Here is what a process block looks like:



An **input/output block** is shaped like a parallelogram. These blocks are used whenever an algorithm requires an input or produces an output. Similar to the process block, this block will have an arrow leading to it from a previous block, and an arrow leading out of it to the next block. Here is what an input/output block looks like:



Here is a possible flowchart that represents the solution to the *get dressed* algorithm:



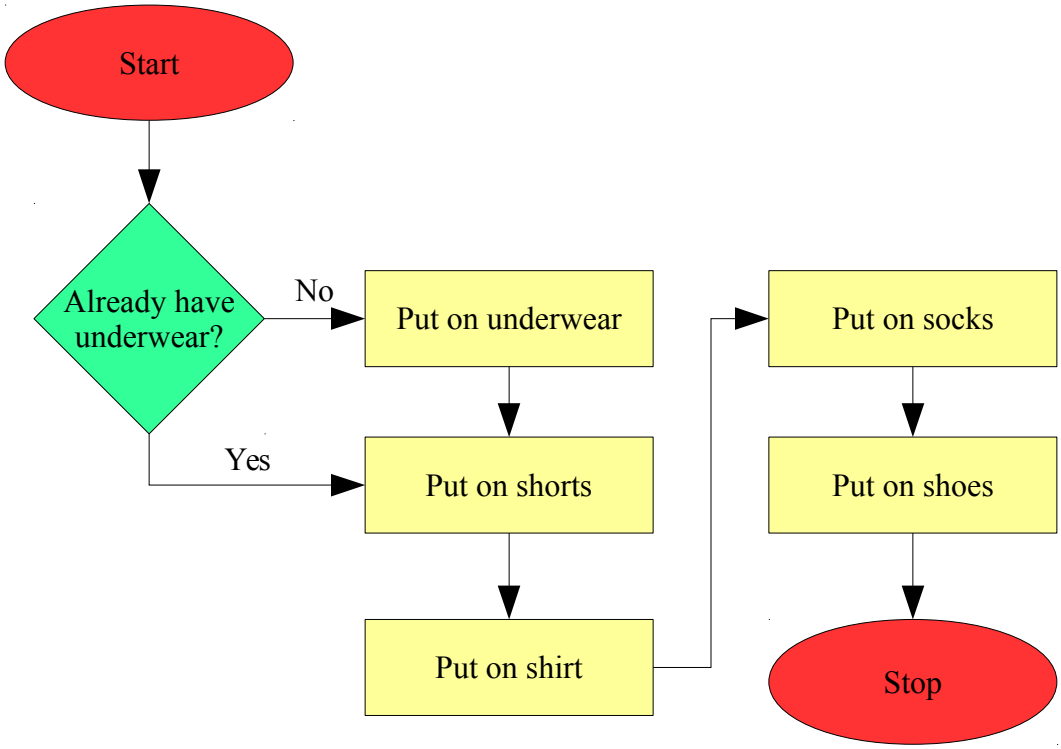
Are there any weaknesses with this algorithm? Will it always work and produce the desired output? For example, will it work if someone already has underwear on? The current algorithm would force the person already wearing underwear to put on a second one. What would happen if the shirt they found was dirty, but they wanted to wear clean clothes instead? This algorithm would force the person to wear the dirty clothes.

These are examples of scenarios that call for *decisions* to be made and instructions to be decided on that are executed based on answers to simple questions. For example, skipping the *put on underwear* step would be useful if the person already had underwear on. Perhaps adding a step *find clean shirt* to the algorithm would handle the case that an original shirt is dirty.

It is difficult to handle these decisions in a traditional to-do list, but flowcharts deal with them in a pretty neat way by using **decision blocks**. Decision blocks are diamond-shaped and typically contain a question with a *yes* or *no* answer. Similar to the previous blocks, the decision block will have one arrow leading into it from the preceding block. However, decision blocks can have two arrows leading out that

go to two different blocks. One of those blocks is executed if the answer to the question posed in the decision block is *yes*, and the other block is executed if the answer is *no*. Here is what a decision block looks like:

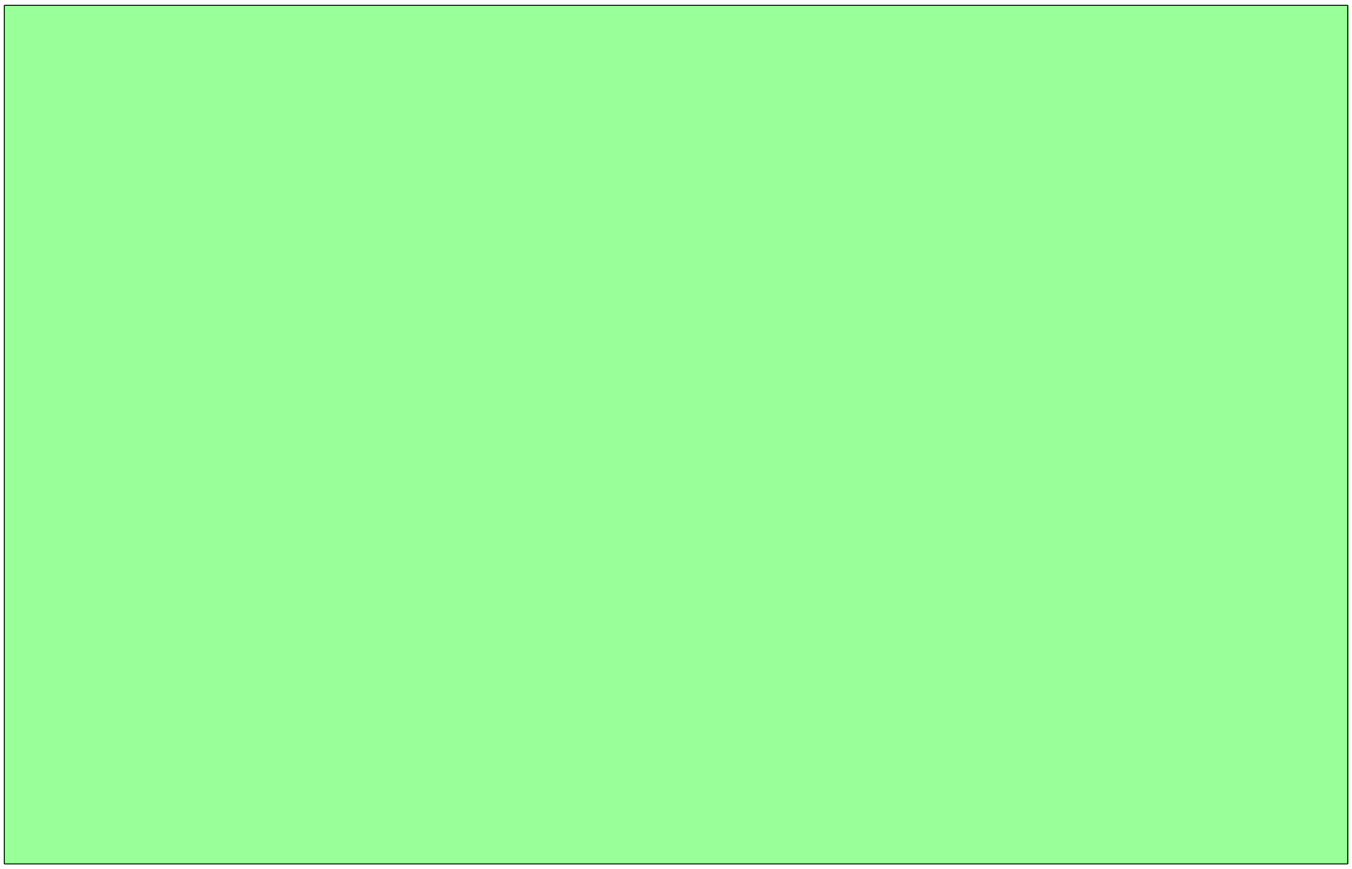
Let's look at an improved flowchart for the *get dressed* algorithm that has a decision block. A possible solution is:



One of the great things about decision blocks (and flowcharts in general) is that you can have different control flows based on decisions made by the algorithm. The algorithm doesn't always have to behave the same way and can actually change its behavior based on what is happening during its execution. The algorithm designer, however, has to think up all possible scenarios and cater for them when designing an algorithm.

Activity 1

Design a flowchart for the *eat breakfast* algorithm. This time, add a decision block with the question *am I satisfied?* in an appropriate position, such that the algorithm will always ensure that the person eating breakfast only stops eating once satisfied.



Repetition
You may have observed from the *eat breakfast* flowchart that it is possible to use decision blocks to repeat a task an indefinite number of times. Repetition is a feature of many algorithms. Repetition is sometimes referred to as **iterating**, and each loop that is repeated is referred to as an **iteration**. Sometimes an algorithm designer will know how many times to repeat a certain task (i.e., how many iterations are needed). For this reason, there are many different types of repetition. Some repeat while a condition exists (we formally call this a **while** loop); some repeat until a condition exists (we formally call this a **repeat-until** loop); some repeat at least once, while a condition exists (we formally call this a **do-while** loop); and some repeat a fixed or known number of times (we formally call this a **repeat-n** loop or a **for** loop, depending on the programming language used). Flowcharts make the process of repetition easy because connecting arrows can easily be placed to go back to an earlier task.

For the next task, let's design an algorithm that finds and displays all the prime numbers that are less than 10. Listing all of the prime numbers that are less than 10 is a fairly easy task. However, what about listing all the prime numbers that are less than 100? What about those that are less than 1,000? Or 10,000? As the number gets larger, it becomes much more complicated. This is why designing an algorithm that a computer could execute would be beneficial. Such an algorithm would calculate all of the required prime numbers faster than we could calculate them on our own.

Did you know?

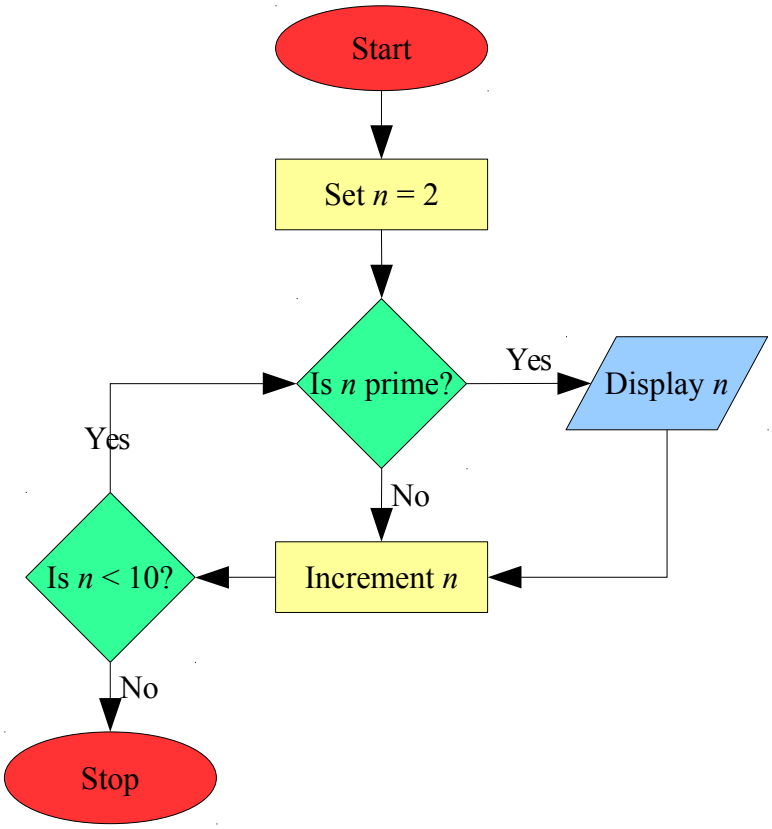
In algorithms where a certain task is repeated, it is often necessary to keep track of how many times it has been completed. This is done by declaring and updating what is referred to as a **variable**. Think of

a variable as a *container* with a name in which we store any piece of data, potentially change it, observe it, and/or compare it with another piece of data as the algorithm is executed.

In the case of repetition, a variable is typically used to represent a number showing how many times a task has been completed. The variable can be called any name we want (the name could be a single letter such as *x* or *n*, or even a long word such as *theVariableStoringTheValueOfOurCounter*). The variable is typically initialized to 0 and is increased by 1 every time a task is completed. When the entire algorithm terminates, the variable will contain the total number of times the task was completed.

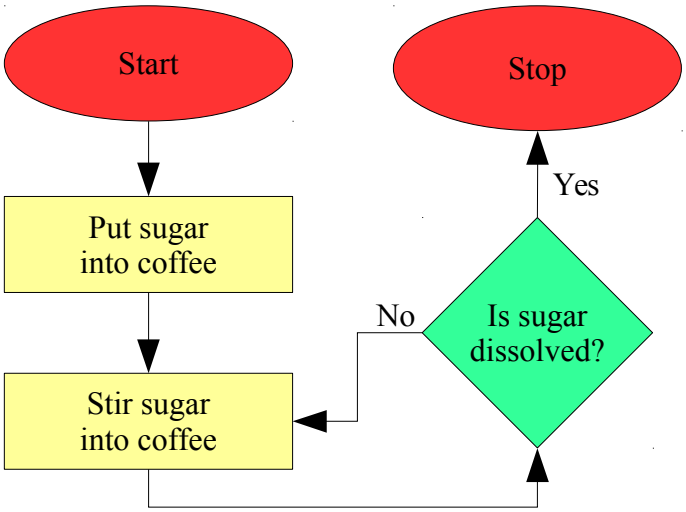
The design of an algorithm that displays all of the prime numbers that are less than 10 requires defining what it means for a number to be prime. Formally, a number is prime if it is only evenly divisible by one and itself. For example, 5 is prime but 4 is not (since it is also evenly divisible by 2). The only even prime number is 2.

To simplify the process of designing the algorithm, let's define a variable, *n*, to be used to specify the current prime number candidate. Let's also define a *magic* step (called *is n prime*) that will be used to determine whether *n* is prime (i.e., let's not do this manually but assume that the question can be asked and an answer provided in a single step). A possible algorithm as a flowchart is:



Notice that in the algorithm above, *is n prime* is repeated for every number (2 through 9 – which is a total of 8 times). If the conditions of the algorithm were changed to find all of the prime numbers below 100, *is n prime* would be repeated for every number from 2 through 99 (a total of 98 times). This algorithm isn't the most *efficient* algorithm, but it gets the job done.

Sometimes the algorithm designer does not know how many times a process is going to be repeated. In such cases, a condition that is needed to be satisfied before the repetition is ended will have to be specified. The statement would then be repeated until that condition is reached. For example, what would an algorithm for stirring sugar into a cup of coffee look like? Would it require stirring five times? 10 times? 100 times? There is no way to tell ahead of time how many times one would have to stir to get the sugar dissolved. However, we know that the stirring should go on while the sugar is not completely dissolved. A possible solution to this is:



Efficiency and runtime

It is sometimes useful to measure and predict the time a computer takes to execute an algorithm. This can help us to compare several solutions to a problem and pick the fastest one, for example.

Definition: ***Runtime*** *is the time that a computer takes to execute an algorithm.*

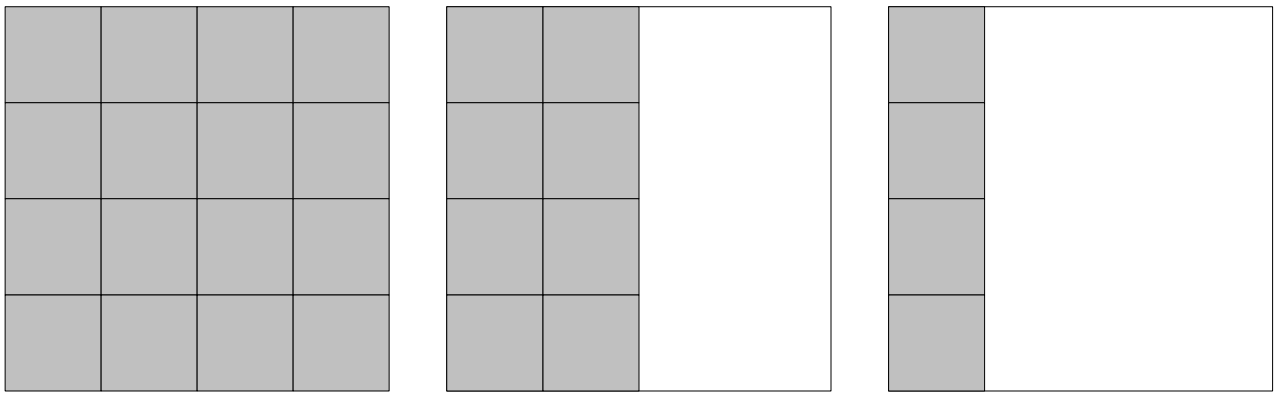
Most of the algorithms that we have previously considered execute all of their statements once. As a result, they would not take long to be executed by a typical computer (i.e., their runtime would be low). However, algorithms that require repetition have a longer runtime. Consider the *find all prime numbers* algorithm. If it was adjusted to find all the prime numbers below 100, it would take considerably longer than if it was set to find all the prime numbers below 10. It would take even longer to find all the prime numbers below 1,000. This is because the *check if prime* process is evaluated for every prime number candidate, and each time it is evaluated the runtime increases.

It is typically desired to design algorithms that have as short of a runtime as possible. Sometimes this calls for designing more intricate or complicated ones; however, only as long as we are sure that the complicated algorithm will have a shorter runtime than the basic algorithm.

Let's take a look at an interesting problem that will help to show the differences between an algorithm that works and an algorithm that works *and is efficient*. Consider a rectangular room and an unlimited number of identical *square* tiles. Can an algorithm be designed to calculate the number of tiles required to cover the entire floor of the room?

There are several ways that this algorithm can be designed. One approach is to lay down tiles in the entire room and then count them. Another approach may be to lay down tiles in half of the room, count the number of tiles used, and then double that number. A third approach may be to lay tiles along one

edge of the floor, and multiply the number of tiles laid by itself (i.e., squaring it to find the *area* of the floor). The following figure shows the three methods, side-by-side.



Which solution do you think is the best? What does *best* mean? Does it mean that the algorithm takes less time? Does it mean that it requires fewer tiles? Does it mean that it requires fewer calculations (and is perhaps less prone to arithmetic errors)? Is there a solution that does not require laying down any tiles at all?

Having multiple solutions (or algorithms) to the same problem is a frequent scenario in computer science. We have mentioned before that *best* is a very subjective measure for an algorithm. However, there is still a need to compare algorithms and determine which algorithm is better. One of the ways of **quantitatively** (i.e., numerically) comparing is by using the algorithm's runtime. Let's take a closer look at some algorithms that solve the tile laying problem. For simplicity, we will assume that both the room and tiles are **square** in shape (i.e., the number of tiles required to cover adjacent edges is equal). Note that the algorithm steps are numbered, with sub-steps placed within a single algorithm.

Algorithm 1 (this one covers the entire floor with tiles):

```
1: set number of tiles currently laid to 0
2: repeat the following steps until the entire floor is covered
     2.1: lay a tile on the floor
     2.2: add one to the number of tiles currently laid
3: the number of tiles currently laid is the number of tiles needed
```

Algorithm 2 (this one covers half of the floor with tiles):

```
1: set number of tiles currently laid to 0
2: repeat the following steps until half of the floor is covered
     2.1: lay a tile on the floor
     2.2: add one to the number of tiles currently laid
3: multiply the number of tiles currently laid by 2
4: the result is the number of tiles needed
```

Algorithm 3 (this one covers the length of one wall with tiles):

```
1: set number of tiles currently laid to 0
2: repeat the following steps until one row has been laid
     2.1: lay a tile on the floor
     2.2: add one to the number of tiles currently laid
3: multiply the number of tiles currently laid by itself
4: the result is the number of tiles needed
```

We are now going to figure out which algorithm is better between Algorithm 1 and Algorithm 3 using their runtime. Suppose that the room is 12ft x 12ft and each tile is 1ft x 1ft. Also assume that it takes 10 seconds to lay each tile. How long will Algorithm 1 take to be completely executed? Since Algorithm 1 calls for tiles to be laid across the entire room, the 12ft x 12ft room would then require 144 tiles. Since it takes 10 seconds to lay each tile, it would then take 1,440 seconds to cover the entire room. This is 24 minutes:

$$1{,}440\,sec * \frac{1\,min}{60\,sec} = 24\,min$$

Did you know?

Dimensional analysis is a nice way to work through problems with different units (like minutes and seconds), and that require conversion across them. The basic idea is that values can be multiplied by conversion (or dimensional) units that are expressed as fractions. Those units can be canceled out if they appear in both the numerator and denominator. The example above can be rewritten as follows:

$$\frac{1{,}440\,sec}{1} * \frac{1\,min}{60\,sec} = 24\,min$$

The *sec* units in the numerator of the first fraction and the denominator of the second fraction cancel each other out, thereby leaving *min* as the final unit. Here's another example that converts days to seconds:

$$\frac{1\,day}{1} * \frac{24\,hr}{1\,day} * \frac{60\,min}{1\,hr} * \frac{60\,sec}{1\,min} = 86{,}400\,sec$$

The *day*, *hr*, and *min* units in the numerator and denominator of the fractions cancel each other out. The only unit left is *sec* which, once the numerators and denominators are multiplied, represent the number of seconds in one day.

What about Algorithm 3? This algorithm only calls for laying down tiles along one edge of the room to make one row. This means that only 12 tiles will be laid down using this algorithm, a process that would take 120 sec (2 min). But Algorithm 3 also requires a calculation. Let's assume that this calculation takes 60 sec. The total runtime for Algorithm 3 is then 120 sec + 60 sec = 180 sec (3 min).

So Algorithm 3 is 21 minutes faster than Algorithm 1 for a 12ft x 12ft room. But what happens if the size of the room is changed? What is the performance of both algorithms if they were used to tile a room that is 20ft x 20ft, under the same timing assumptions as before?

We find that Algorithm 1 takes approximately 1 hr 7 min:

$$\frac{400\,tiles}{1} * \frac{10\,sec}{1\,tile} = 4{,}000\,sec$$

$$\frac{4{,}000\,sec}{1} * \frac{1\,min}{60\,sec} * \frac{1\,hr}{60\,min} = 1.11\,hr$$

$$\frac{0.11\,hr}{1} * \frac{60\,min}{1\,hr} = 6.6\,min$$

$$\frac{0.6\,min}{1} * \frac{60\,sec}{1\,min} = 36\,sec$$

Let's explain the calculations above. The first calculates the number of seconds required for Algorithm 1: 4,000 sec. To represent this in hr, min, and sec, we simply need to convert 4,000 sec to hr (the second calculation): 1.11 hr. The third calculation takes the fractional portion of this (0.11 hr) and converts it to min: 6.6 min. The final calculation takes the fractional portion of this (0.6 min) and converts it to sec: 36 sec. The total time is then 1 hr 6 min 36 sec, or approximately 1 hr 7 min.

Algorithm 3 only takes approximately 4.5 minutes:

$$\frac{20\,tiles}{1} * \frac{10\,sec}{1\,tile} = 200\,sec + 60\,sec = 260\,sec$$

$$\frac{260\,sec}{1} * \frac{1\,min}{60\,sec} = 4.33\,min$$

$$\frac{0.33\,min}{1} * \frac{60\,sec}{1\,min} = 19.8\,sec$$

Again, the first calculation gives us the time it takes to lay tiles across a single wall, and then do the 60 sec calculation: 260 sec. The second calculation converts sec to min. The final calculation takes the fractional portion of this (0.33 min) and converts it to sec: 19.8 sec. The total time is then 4 min 19.8 sec, or approximately 4.5 min.

This means Algorithm 3 is over 15 times faster than Algorithm 1 for a 20ft x 20ft room:

$$\frac{4{,}000\,sec}{260\,sec} = 15.38$$

Does that mean that Algorithm 3 will always be faster and therefore better than Algorithm 1? Try calculating how long it will take for either algorithm to cover a very small room that is 2ft x 2ft and compare them in the space below:

The calculation for Algorithm 1:

The calculation for Algorithm 3:

The runtime comparison:

What about if the room was 3ft x 3ft? In this case, we find that both algorithms take the same amount of time. The calculation for Algorithm 1:

$$\frac{9\,tiles}{1} * \frac{10\,sec}{1\,tile} = 90\,sec$$

The calculation for Algorithm 3:

$$\frac{3\,tiles}{1} * \frac{10\,sec}{1\,tile} = 30\,sec + 60\,sec = 90\,sec$$

The question of which algorithm is better depends on the size of the room to be tiled. Figure 1 compares the runtime of the two algorithms on differently sized rooms. It is a plot in which the horizontal axis represents the length of one wall of the room (in feet), and the vertical axis represents the amount of time that is required to tile the room (in seconds). The figure illustrates that for rooms less than 3ft x 3ft, Algorithm 1 is faster. In the case where the room is exactly 3 ft x 3 ft both algorithms require the same amount of time. For rooms greater than 3ft x 3ft, Algorithm 3 is faster.

Computer programs and pseudocode

So far, we have discussed: (1) how algorithms are step-by-step solutions to specific problems; (2) that algorithms can be represented as to-do lists and flowcharts; and (3) that the runtime of different algorithms for the same problem can be used to compare those algorithms. However, we did not discuss how algorithms are *given* to a computer so that it can execute our algorithms for us.

The main benefit in using computers to execute algorithms is that, typically, they are much faster and can process calculations much more quickly than we can. As humans, we speak and write in different languages (one of which is English). On the other hand, computers can only understand a language that is made up of 1s and 0s. We will discuss the details of this **machine language** later on.

Therefore, there is a need to represent algorithms in a form (or language) that is easy for both a human being and a computer to understand. This helps us to specify algorithms correctly while also allowing computers to execute them quickly. The language that we use to do this is called a **programming**

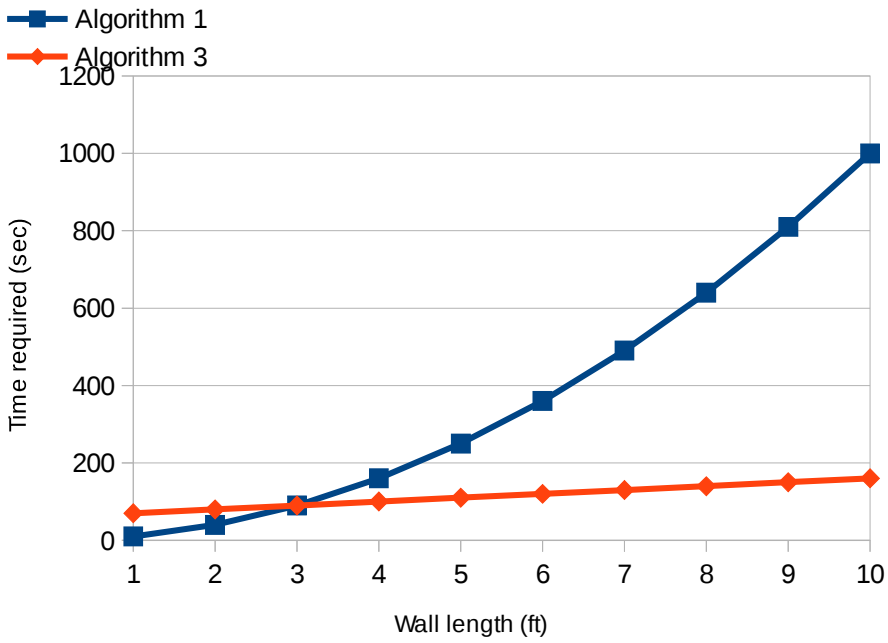


Figure 1: Algorithm performance on rooms of various sizes

language. There are many programming languages, some of which you will learn to use in this curriculum. They usually share some similarities with English, and this is what makes it easier to describe our algorithms instead of trying to *speak* 1s and 0s. When we have represented our algorithm in a programming language, it is referred to as a **computer program**.

Definition: *A **programming language** is a language designed to communicate instructions to a computer.*

It is important to remember that a computer will only blindly follow the instructions that it is given by a human being. It never actually understands the problem that it is solving. It is the responsibility of the programmer to understand the problem and create an algorithm that solves that problem. Good algorithms require creativity, imagination, and a lot of hard work.

The computer programming process can be compared to composing a piece of music. Composing a piece of music takes creativity. The composer then has to transfer the music from his head to a piece of paper in a way that other people can understand and perform it (e.g., music scales). The composer imagines the piece of music and then encodes it as a series of symbols on paper. This process requires skill and creativity. The computer executing a program can be compared to performing the piece of music. The performer goes through a series of mechanical motions as described by the musical scales (or instructions) that were designed by the composer.

Our job as computer scientists is to become very good composers. We will need to learn how to transfer the beautiful ideas that we have in our heads into a series of instructions that a computer can blindly follow to make our *music*.

One more way of representing algorithms is called **pseudocode**. This way of representing algorithms was actually shown earlier in this lesson when the tile laying algorithms were described. Pseudocode is

a way of writing algorithms in a manner that shares similarities with programming languages. It is a mixture of normal English and programming language concepts. This method of representing algorithms will be used more often than the others in the curriculum. Here is an example of the pseudocode for an algorithm that models the *stirring of sugar into coffee*:

```
1: put sugar into coffee
2: repeat
3:     stir sugar into coffee
4: until sugar is dissolved
5: stop
```

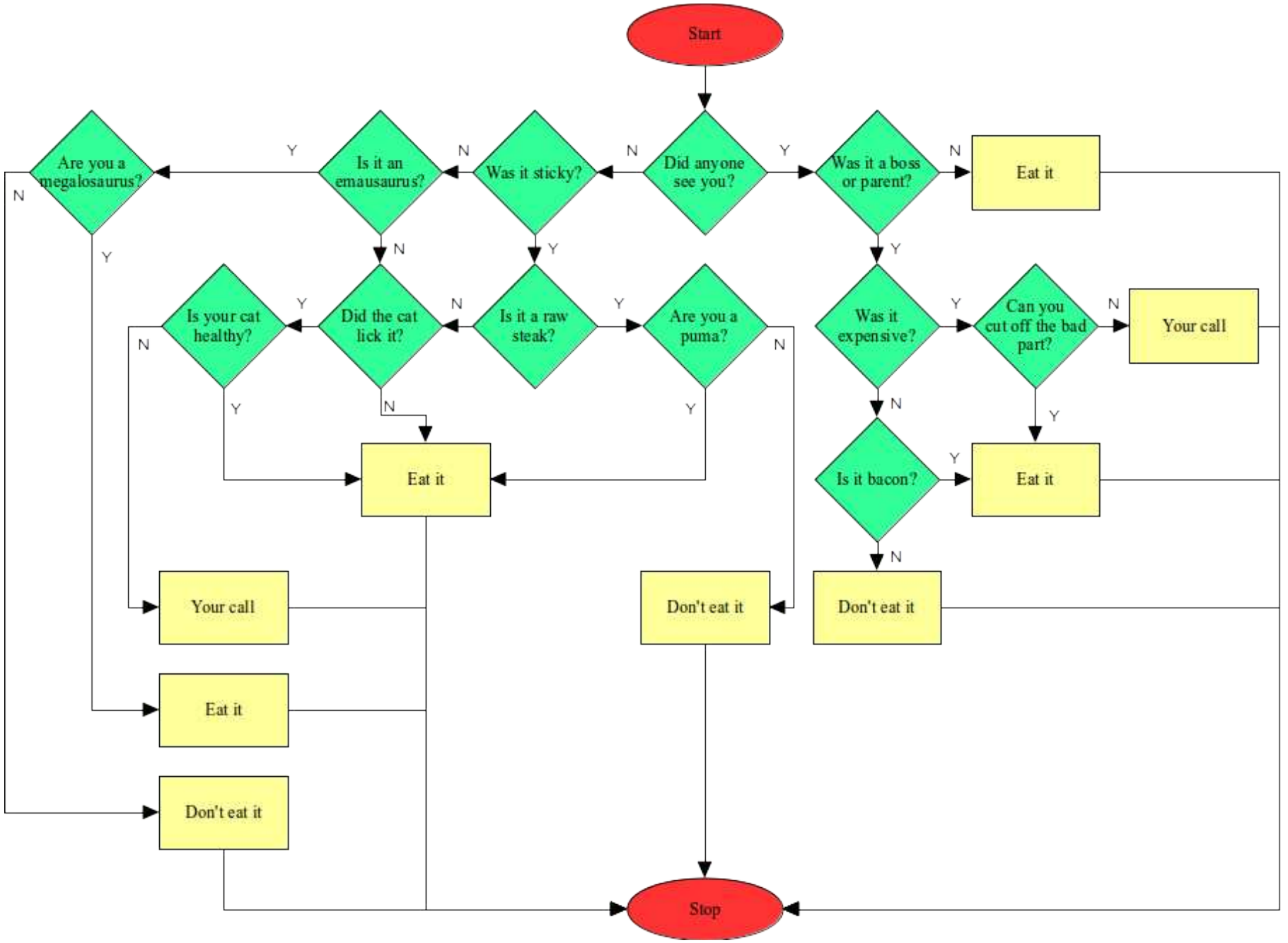
Here is the pseudocode for displaying *all prime numbers less than 10*:

```
1: n ← 1
2: repeat
3:     if n is prime
4:     then
5:         display the value of n
6:     end
7:     increase n by 1
8: until n >= 10
9: stop
```

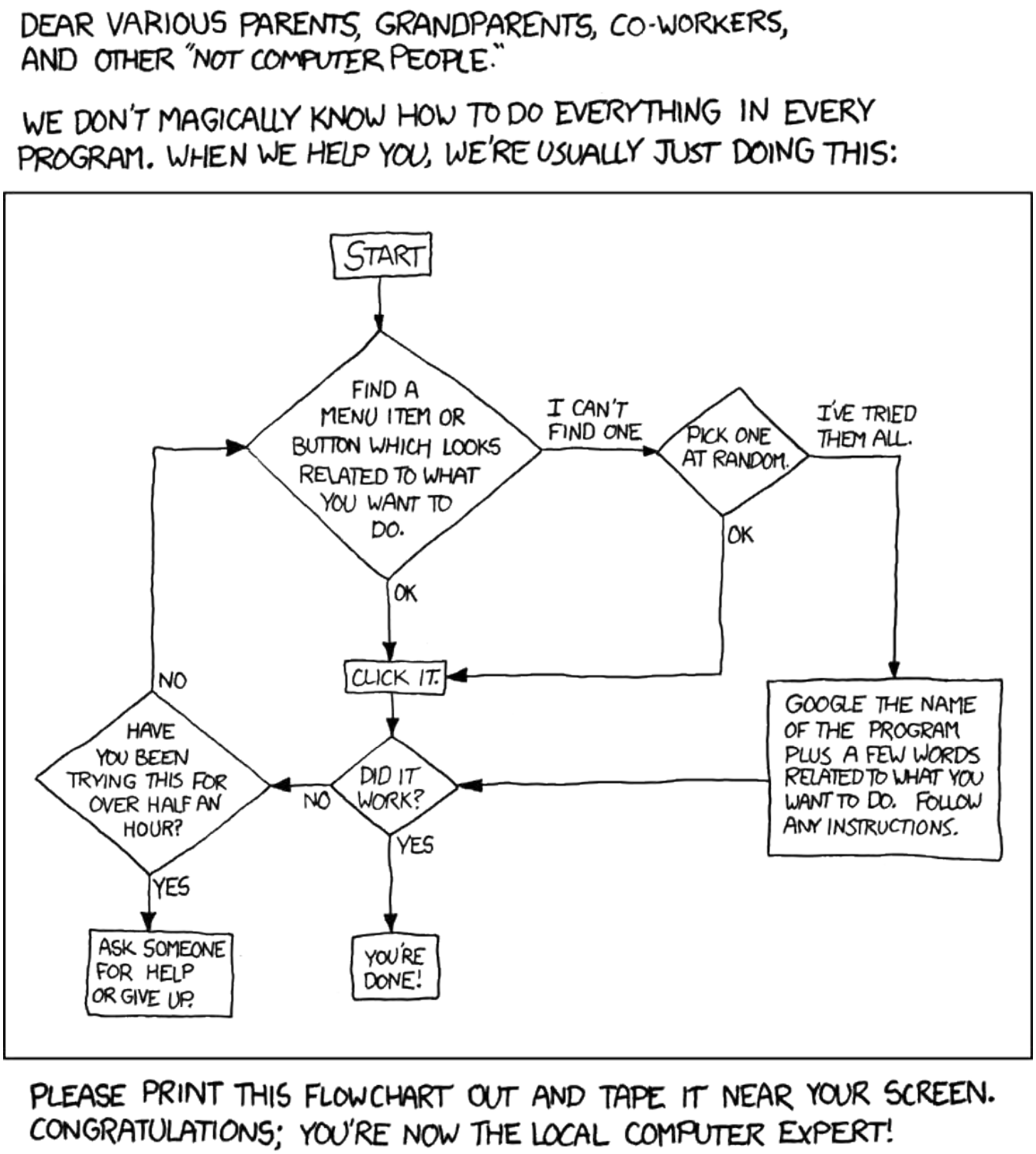
Note how **keywords** involving major concepts (e.g., decision making, repetition) are highlighted in bold. Indentation is used to identify grouped instructions, while the numbers on the left become line numbers that make it easier to refer to particular lines in the pseudocode. Finally, assignment of values to variables is done with a left arrow.

A tangled web

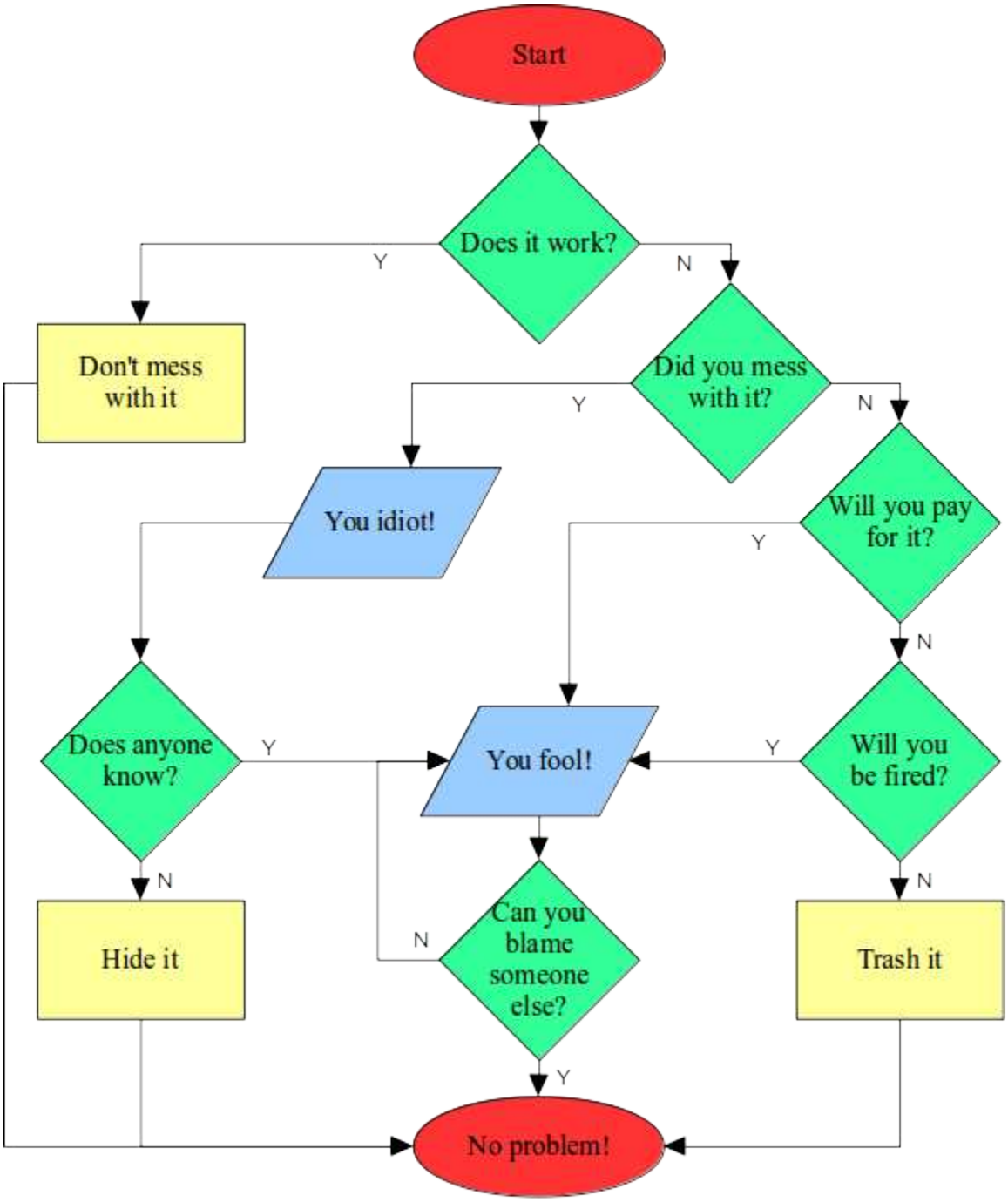
Can you determine what the flowcharts below do? Try them out with different scenarios.


Start
Did anyone see you?
Was it sticky?
Is it an emausaurus?
Are you a megalosaurus?
Was it a boss or parent?
Eat it
Is it a raw steak?
Did the cat lick it?
Is your cat healthy?
Are you a puma?
Was it expensive?
Can you cut off the bad part?
Your call
Is it bacon?
Eat it
Eat it
Your call
Don't eat it
Don't eat it
Eat it
Don't eat it
Stop
Y
N

Flowcharts are often used to make a point. Take, for example, the following flowchart that could express a computer scientist's frustration (from XKCD):



Here's another funny one?



Machine language

Definition: ***Machine language*** *is a set of specific instructions that can be executed directly by a computer. This language is typically made up of binary digits (1s and 0s).*

A computer can only understand machine language. Machine language instructions are entirely made up of binary digits and can be directly executed by the CPU. Since we have a particularly hard time understanding 1s and 0s, early programmers assigned a set of mnemonics to represent machine code instructions so that they would be a bit more readable. This mapping became known as **assembly language**. People still write code in assembly language, but it is not typically used to create large scale applications.

Programming language

The kinds of languages that are widely used today are known as programming languages. Programming languages allow us to represent algorithms in a way that is similar to English but is more structured and much less ambiguous.

Definition*: A* ***programming language*** *is a precisely constructed language that is specifically used to communicate instructions to a computer.*

English is a spoken language. As such, it was spoken first, rules were later defined and written down. Therefore, there are many exceptions (i.e., words and phrases that are grammatically correct but don't conform to the general grammar/spelling rules). Spoken languages are sometimes ambiguous and open to interpretation. This means that a single statement can have multiple meanings. For example, the statement "I made the robot fast" can mean several different things. Does it mean that the robot was built quickly? Or does it mean that the robot was modified so that it would move around more quickly than it did before? Perhaps it means that the robot is named Fast. Or maybe that we managed to make the robot stop eating nuts and bolts.

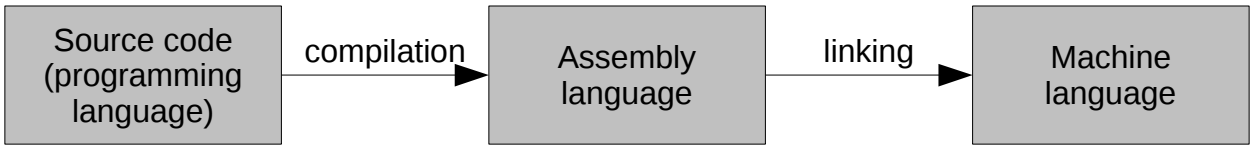
Humans rely on external factors like context and body language to understand the true meaning of a statement in a spoken language. And even then mistakes in interpretation still happen. With computers however, we need a language that is so structured and unambiguous that every computer can understand and interpret a given statement in the exact same way. For example, we don't want two different computers giving us two completely different answers to the arithmetic expression `1 + 1`.

In contrast to spoken languages, programming languages are first defined with rules. The language itself is then derived from those rules. Programming languages are therefore quite structured and not ambiguous. They are very precise and logical.

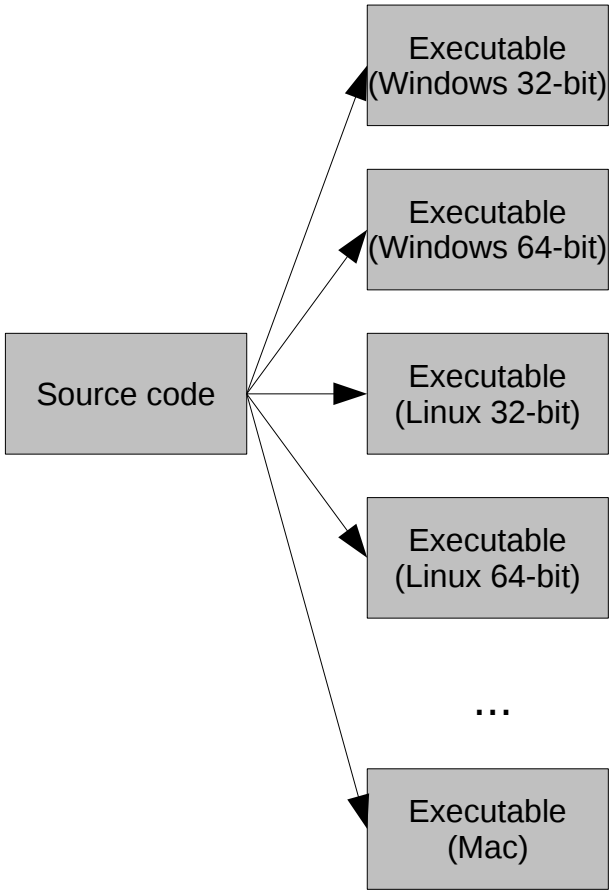
There are many different programming languages that can be used to describe an algorithm. One of them, for example, is called Python, and it is what we will be using for the majority of the Living *with* Cyber curriculum. It is the duty of the programmer to write down the tasks that he/she wants done in a given programming language. Since computers can only understand machine language, we utilize an application known as a **compiler** that translates this programming language into machine language.

***Definition**: A **compiler** is a tool used to translate an algorithm expressed in a programming language to machine language. The process by which this conversion from programming language to machine language is done is called **compilation**.*

The compilation process takes an algorithm written in a programming language and translates it to assembly language. From there, a process known as *linking* converts the assembly language to machine language. This is illustrated by the figure below:



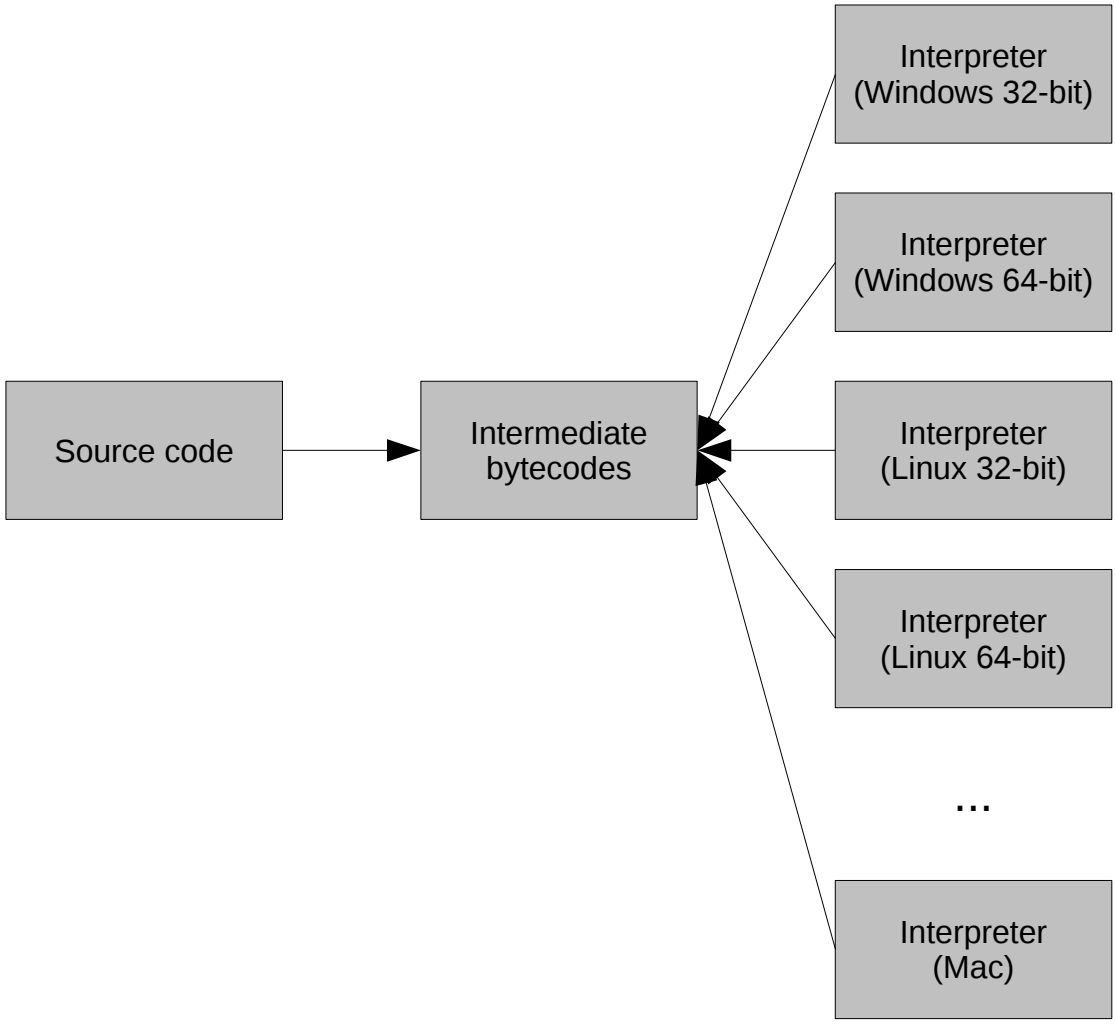
Once machine language is generated to match a program, the computer can then directly execute the program and implement the algorithm. A fully compiled language is only executable by a CPU with the same characteristics and operating system (often, including version) as that which it was compiled for. A programmer who wants wide distribution of his software will need to compile source code to the various destination computing architectures and operating systems that are the most likely to be used by the target audience for the application. Of course, the programmer could simply distribute source code and let users compile that themselves. Often, however, programmers do not wish to distribute source code for a variety of reasons (e.g., intellectual property). The figure below shows how a program would need to be compiled numerous times to cover a range of target computing architectures and operating systems:



Not all programming languages are compiled to machine language. Some are never compiled and are executed, one instruction at a time, by an **interpreter**. An interpreter can be thought of as a real time compiler that executes high level programming language instructions, one at a time. Interpreted languages are much slower to execute than compiled languages. Examples of interpreted languages are Python, PHP, JavaScript, and Perl. To execute a program written in an interpreted language, you must have an appropriate interpreter installed on your computing system.

Interpreted languages also require programmers to distribute their source code, and users to have an appropriate interpreter installed on their system. Maintaining code privacy is therefore not possible with interpreted languages.

Partially compiled and interpreted languages combine the convenience of interpreted languages (i.e., not having to compile source code to a large number of target machine language executables) and the privacy and speed of compiled languages (i.e., not having to distribute source code). These types of programming languages are partially compiled (to an intermediate language) and then interpreted from there. Examples of partially compiled languages are Java, Python, and Lisp. Note that Python can be strictly interpreted or partially compiled depending on the programmer's preferences. The intermediate language is distributed and subsequently executed on any computing platform that has an interpreter for the intermediate language. For example, Java source code is typically expressed in a `.java` file and partially compiled to Java bytecodes (in a `.class` file) that can then be distributed. A Java Virtual Machine (JVM) executes the bytecodes by interpreting each instruction, one at a time. The benefit of this method is that a programmer can distribute a single file to everyone, regardless of CPU architecture and operating system. Anyone wanting to execute the file simply needs to have a version of the JVM for their computing system. this is illustrated in the figure below:



Programming paradigms
Over the past forty years or so, three general classes, or paradigms, of programming languages have emerged. These paradigms include the imperative paradigm, the functional paradigm, and the logical paradigm. In addition, during the past decade or so these paradigms have been extended to include object-oriented features. A language is classified as belonging to a particular paradigm based on the programming features it supports.

Object-oriented imperative languages are, by far, the most popular type of programming language. Both Java and C++ (two of the most used programming languages in industry) are object-oriented imperative languages. Scratch and Python are imperative languages – although Python does contains object-oriented attributes, Scratch does not.

The **imperative paradigm** is based on the idea that a program is sequence of commands or instructions (usually called **statements**) that the computer is to follow to complete a task. The imperative style of programming is the oldest, and now with object-oriented extensions, continues to be far and away the most popular style of programming.

The Living *with* Cyber curriculum first (and very briefly) utilizes Scratch as the programming language. This is quickly followed by Python. Scratch is not intended to be used to create applications designed

for production systems. That is, it is not a *general purpose* programming language. Instead, it is a teaching tool aimed at simplifying the process of learning to program. Scratch purposefully omits many features available in other popular programming languages in order to keep the language from becoming overly complex. This allows you to focus on the *big picture* rather than get bogged down in the complexities inherent in *real* programming languages and their development environments.

One way of thinking about writing Scratch programs is to compare it to programming in a *production* programming language with training wheels on. Complex and useful programs can be written in Scratch; however, there are many things that programmers are allowed to do in production languages that are not possible (at least not straightforward) in Scratch. For example, Scratch does not support functions and function calls directly, nor does it support recursion directly. These terms may not be familiar right now; however, these restrictions are designed to help beginning programmers avoid making common mistakes.

General purpose programming languages are more robust, and can (and are) used in more situations than educational programming languages like Scratch. Think of it like this: using a programming language like Scratch is like building a Lego house only using 2x4 Lego pieces. While it is possible to do so, there is a limitation on what kinds of houses you can build. Conversely, using more general purpose programming languages is like building a house with any kind of Lego piece you can think up in your mind. There are fewer limitations, and the kinds of houses that you can build are limitless. From this point, we will use Python as the general purpose programming language in the course.

Why Python?
You may have heard about other general purpose programming languages: Java, C, C++, C#, Visual Basic, and so on. So why use Python instead of, say, Java? In the end, it amounts to the simple idea that, unlike all of the other general purpose programming languages listed above, Python allows us to create powerful programs with limited knowledge about syntax, therefore allowing us to focus on problem solving instead. In a sense, Python is logical. That is, nothing must be initially taken on faith (that will ostensibly be explained at a later time). There isn't any excess baggage that's required in order to begin to write even simple Python programs.

Recall how, in geometry, the formula for calculating the volume of a cone was given. At that time, it was simply inexplicable. That is, you were most likely told to memorize it. It is not until a calculus course that this formula is actually derived, and how it came to be is fully explained. Why? Well, it is simply because it requires calculus in order to do so. Most students taking a geometry course have not yet had calculus; however, formulas for calculating the volume of various objects (including a cone) are typical in such a course. The problem, of course, is that we are told to take it on faith that it, in fact, works as described. We are told that, how it works and how it was derived, will be explained at a later time. The problem with this is that it forces memorization of important material as opposed to a deep understanding of it (which, in the end, is the goal).

A similar thing actually occurs in a lot of programming languages. Often, we must memorize syntax that will be explained later. Python is unique in that it does a pretty good job of taking all of that out by just being simple. Programming in Python is immediately logical and explicable.

Take the following simple example of a program that displays the text, “Programming rules, man!” in various general purpose programming languages:

In Java:

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

In C:

```
#include <stdio.h>

int main()
{
     printf("Programming rules, man!\n");
}
```

In C++:

```
#include <iostream>
using namespace std;

int main()
{
     cout << "Programming rules, man!" << endl;
}
```

In C#:

```
public class SimpleProgram
{
     public static void Main()
     {
          System.Console.WriteLine("Programming rules, man!");
     }
}
```

In Visual Basic:

```
Module Hello
     Sub Main()
          MsgBox("Programming rules, man!")
     End Sub
End Module
```

And in Python:

```
print "Programming rules, man!"
```

In all of these examples, compiling and running the programs (or interpreting them) produces a single line of output text: “Programming rules, man!” Did you notice that, in all of the examples (except for Python), there seems to be a good bit of seemingly extra stuff for such a simple program? There are a

lot of words that you may not be familiar with or immediately understand: `class`, `public`, `static`, `void`, `main`/`Main`, `#include`, `printf`, `cout`, `namespace`, `String[]`, `endl`, `Module`, `Sub`, `MsgBox`, and so on. In fact, the only readable version to a beginner is usually the one written in Python. It is pretty evident that the statement `print "Programming rules, man!"` means to display that string of characters to the screen (or console).

Python is extremely readable because it has very simple and consistent syntax. This makes it perfect for beginner programmers. It also forces good coding practices and style, something that is very important for beginners (especially when it comes to debugging and/or maintaining programs). Python has a large set of libraries that provide powerful functionality to do just about anything. Libraries allow Python programmers to use all kinds of things that others have created (i.e., we don't have to reinvent the wheel). A huge benefit of Python is that it is platform independent. It doesn't matter what operating system you use, it is supported with minimal setup and configuration, and there is no need to deal with dependencies (i.e., other things that are required in order to just begin to code in Python).

Don't think that, because of its simplicity, Python is therefore not a powerful language (or perhaps that it doesn't compete with Java or C++). Python is indeed powerful, and can do everything that other programming languages can do (e.g., it does support the object-oriented paradigm). It is based on a few profound ideas (collectively known as **The Zen of Python** written by Tim Peters):

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one--obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Did you know?

The name of the Python programming language is taken from a television series called Monty Python's Flying Circus (and not from the snake).

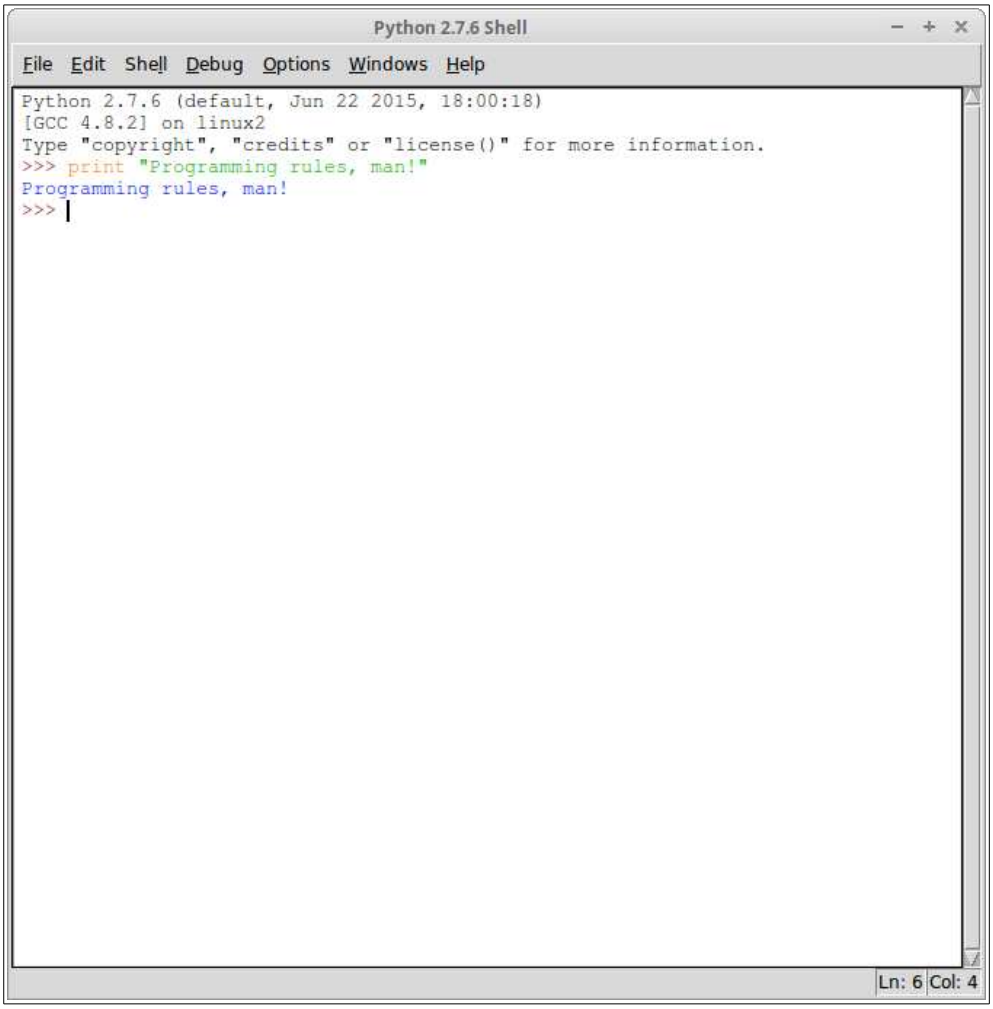
Integrated development environment

Many programmers write their programs written in a general purpose programming language using some sort of text editor (usually a simplistic one, albeit with useful characteristics such as syntax highlighting). In fact, some write programs at the command line (in the terminal) using nothing but a

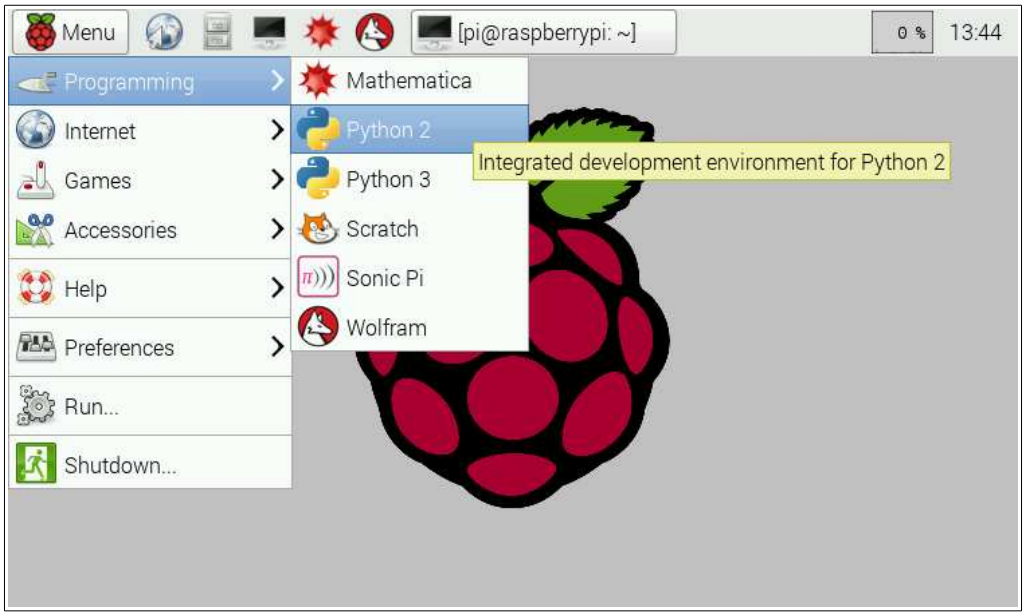
text-based text editor (i.e., without graphical characteristics). Most programmers, however, use an IDE (Integrated Development Environment).

Definition: *An **Integrated Development Environment (IDE)** is a piece of software that allows computer programmers to design, execute, and debug computer programs in an integrated and flexible manner.*

On the Raspberry Pi, the IDE used to design Python programs is called IDLE (which stands for Python's **I**ntegrated **D**eve**L**opment **E**nvironment). Other IDEs exist for pretty much all of the most used general purpose programming languages: Eclipse, Visual Studio, Code::Blocks, NetBeans, Dev-C++, Xcode, and so on. In fact, many of these IDEs support more than one language (some natively, others by installing additional plug-ins or modules)! Here's an image of IDLE with the program shown earlier implemented (and executed):



On the Raspberry Pi, IDLE can be launched as follows:



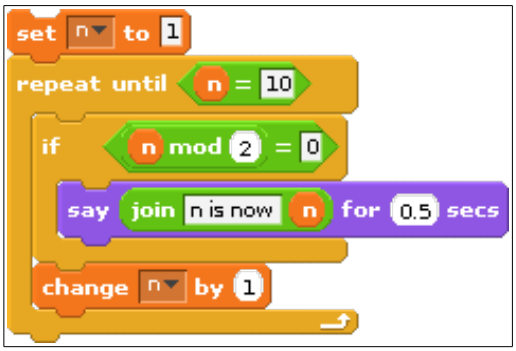
Python programs can also be created and executed at the command line (or terminal). We do so by launching a terminal and typing **python**, which brings up the Python shell:

Scratch vs Python

Earlier in this lesson, you learned that programs written in a programming language are either compiled (to machine language so that a computer can execute them directly) or interpreted, statement-by-statement (in a sense, you could say that programs written in interpreted languages are compiled, line-by-line, in real time). Python is an interpreted language that implements the imperative paradigm. That

is, programs are designed as a sequence of instructions (called statements) that can be followed to complete a task.

Let's take a look at a simple program in Scratch and see how it compares to the same thing in Python:



What does this program do? Simply put, it displays the numbers 2, 4, 6, and 8. Take a look at the script above. The variable *n* is initially set to 1. A *repeat-until* loop is executed so long as *n* is less than 10 (i.e., 1 through 9). Each time the body of the loop is executed, the string "*n* is now (plus the value of *n*)" is displayed if *n* is evenly divisible by 2. For example, if *n* is 4, then the string ***n* is now 4** is displayed. Recall that the **mod** operator returns the remainder of a division. Therefore, when **n mod 2 = 0** is true, it means that the remainder of *n* divided by 2 is zero – so *n* must be even! At the end of the body of the loop, the variable *n* is incremented (ensuring that *n* will eventually reach the value 10, and we will break out of the *repeat-until* loop).

Here's how this can be similarly done in Python:

```
n = 1
while n < 10:
     if n % 2 == 0:
          print "n is now " + str(n)
     n = n + 1
```

At this point, it is fine if you don't understand everything that's going on syntactically. The idea is simply to illustrate how Scratch and Python differ (and are similar!). But let's try to explain. The block, **set *n* to 1**, in Scratch is implemented in Python as, `n = 1`. Pretty similar! Python has no *repeat-until* repetition construct. Instead, we can use a *while* construct with a modified condition. Repeating a task until a variable (in this case, *n*) is 10 is the same thing as repeating it while the variable is less than 10. *If-statements* are similar; however, the **mod** and **equality** operators differ. In Python, we check for equality using the double-equal (==) operator. The mod operator is a percent sign (%). So the block, **if *n* mod 2 = 0**, in Scratch can be implemented in Python as, `if n % 2 == 0`. Generating the output, "*n* is now 4," for example, can be implemented in Scratch using the familiar **print** statement: `print "n is now 4"`. Of course, we don't always want to display that *n* is 4. So we concatenate (or join) the value of *n* to the string "*n* is now " just as we did in Scratch. However, since *n* is not a string of characters (i.e., it is a number – an integer to be precise), then it must first be converted to a string before being concatenated to another string. This is what `str(n)` does. Finally, the value of *n* is incremented by 1 with the statement `n = n + 1`.

In Scratch, it is easy to see the blocks that belong in the body of a repetition construct. The puzzle pieces intrinsically capture this (i.e., they are quite literally visible inside the *repeat-until* block in the

script above). In Python, we denote statement hierarchy (i.e., if statements belong in the body of a construct such as a *while* loop) by using indentation. Note how it is quite clear which statements belong in the body of the *while* loop above: the *if-statement* and the statement that increments the variable *n* by 1. Note that the *print* statement is inside the true part of the *if-statement* (this is evident by how it is directly beneath the *if-statement* and indented further to the right). Again, at this point it is fine to have a minimal grasp of Python's syntax.

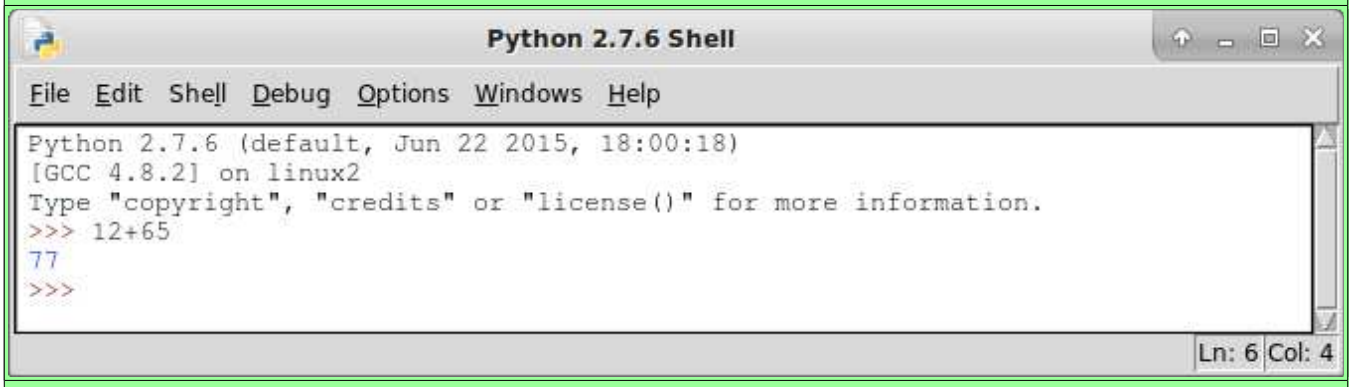
Activity 1: Python Primer

In this activity, you (and/or the prof) will experiment with the Python IDE in order to get a basic understanding of and experience with Python.

First, bring up the Python IDE (IDLE). Formally, we call this the Python shell, an active Python interpreter environment. It's quite useful as it can be used to evaluate expressions in real time (i.e., without saving a program to a file first).

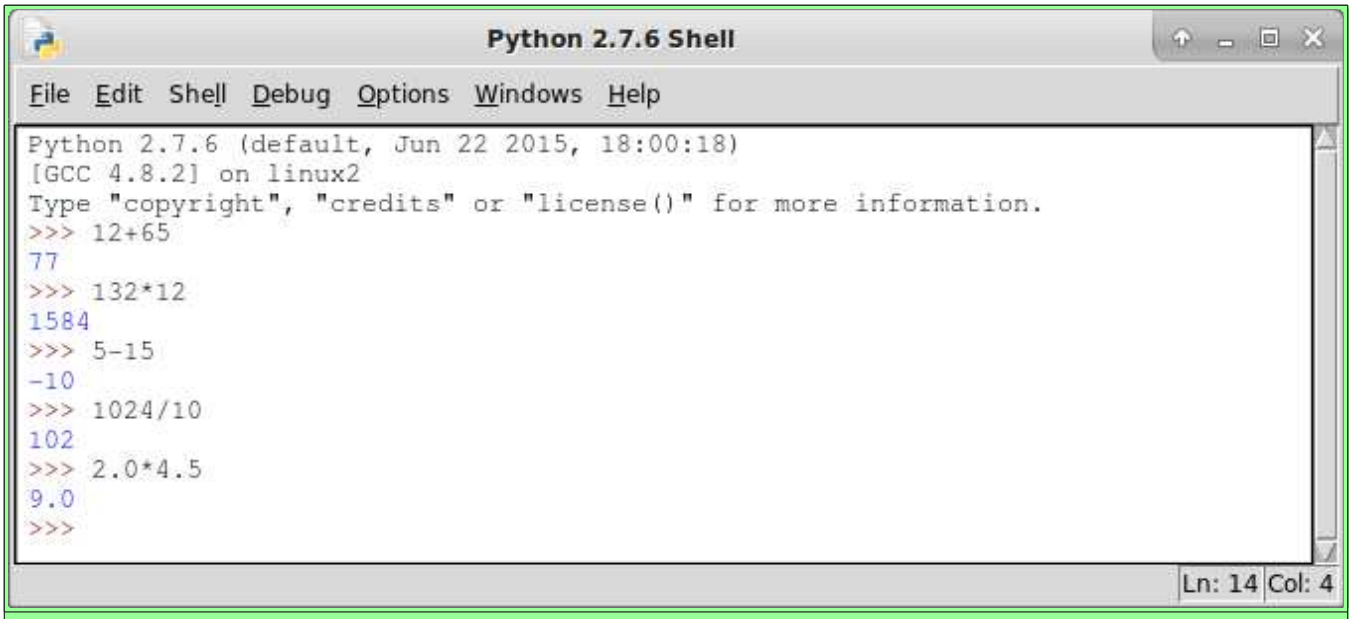
Simple arithmetic expressions

Let's first begin with some simple arithmetic expressions to see how the Python shell can evaluate them and provide real time results. Here's an example of Python evaluating a simple expression (12 + 65) and providing the result in IDLE:



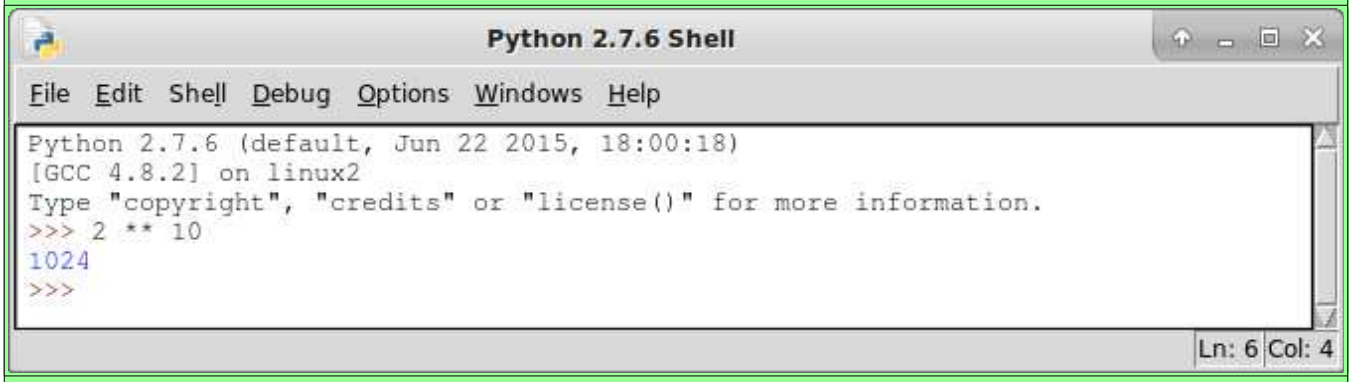
Note that the size of the IDLE window has been reduced in this document.

Python evaluates the expression, 12 + 65, and provides the result in the Python shell. Here are more examples:



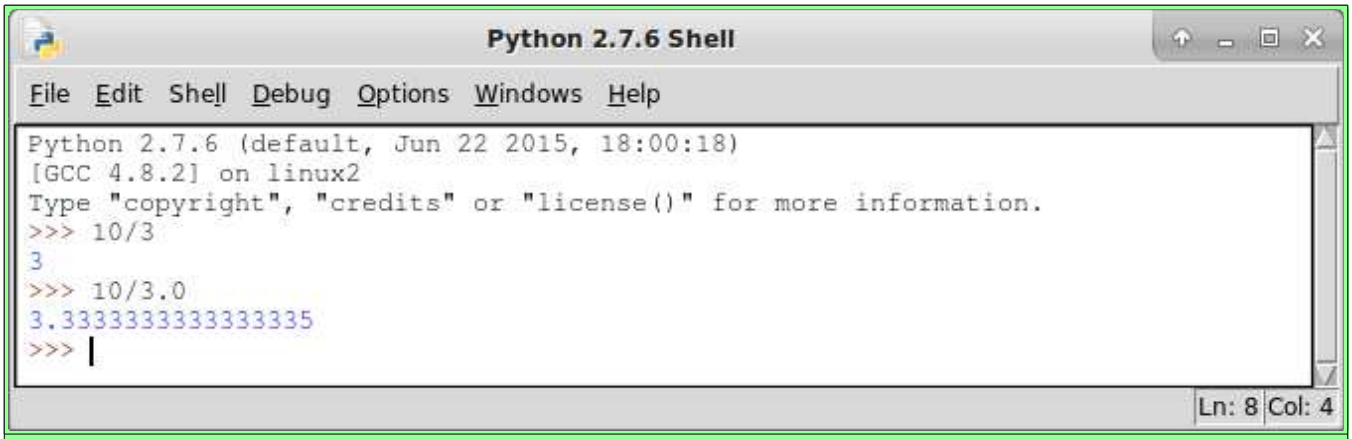
Verify that the expressions are indeed correct (e.g., 5 – 15 = -10, 2.0 * 4.5 = 9.0, etc).

So far, you have seen the four main arithmetic operators (i.e., +, -, *, and /). Take a look at this expression:



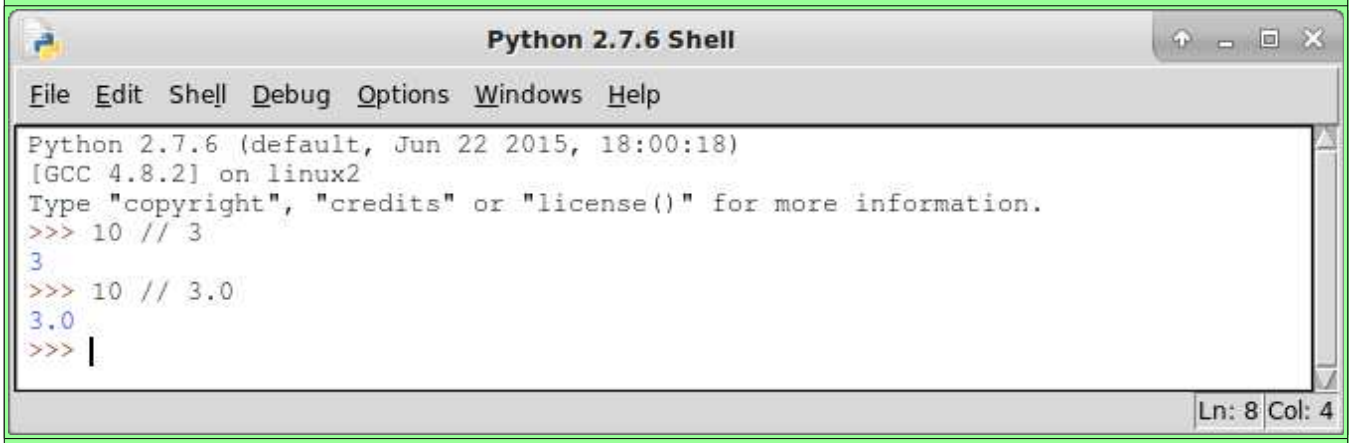
What does the ** operator do? It performs exponential (power) calculation on the two operands. The expression 2 ** 10 implies two raised to the tenth power (or 2^{10}), which is indeed 1024.

Now, take a look at the following expressions:

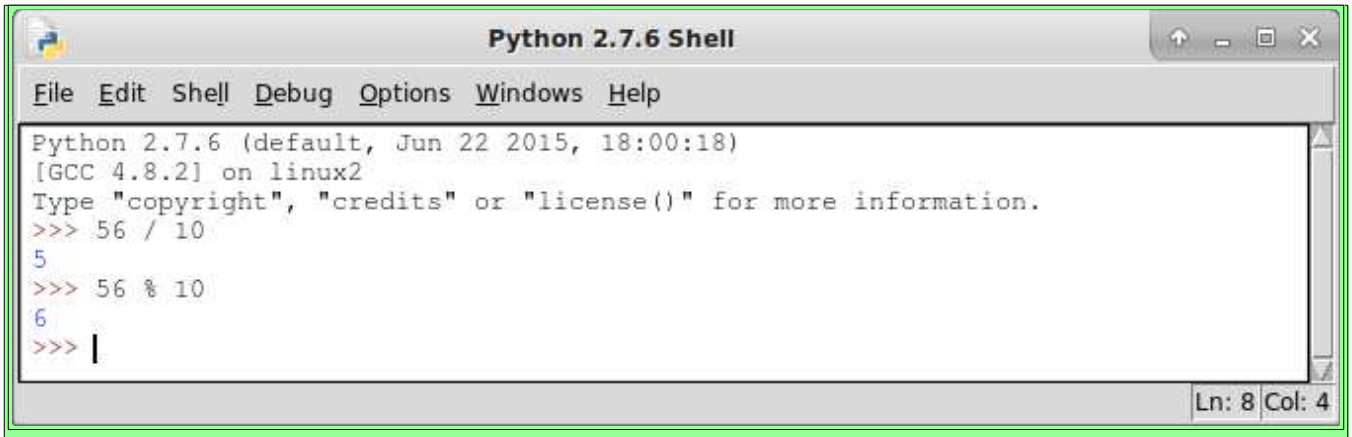


Note how the expression 10 / 3 results in 3. The decimal portion of the result (which we calculate to precisely be 3.33333...) seems to be truncated. In fact, the / operator in Python 2.7.6 returns integer division if the two operands the operator is being applied to are both integers. That is, it returns the integer portion only (i.e., the quotient) of a division of two integers. To perform floating point division, at least one of the operands must then be floating point. This is shown in the second example above.

We can force integer division regardless of the type of operands with the // operator as follows:



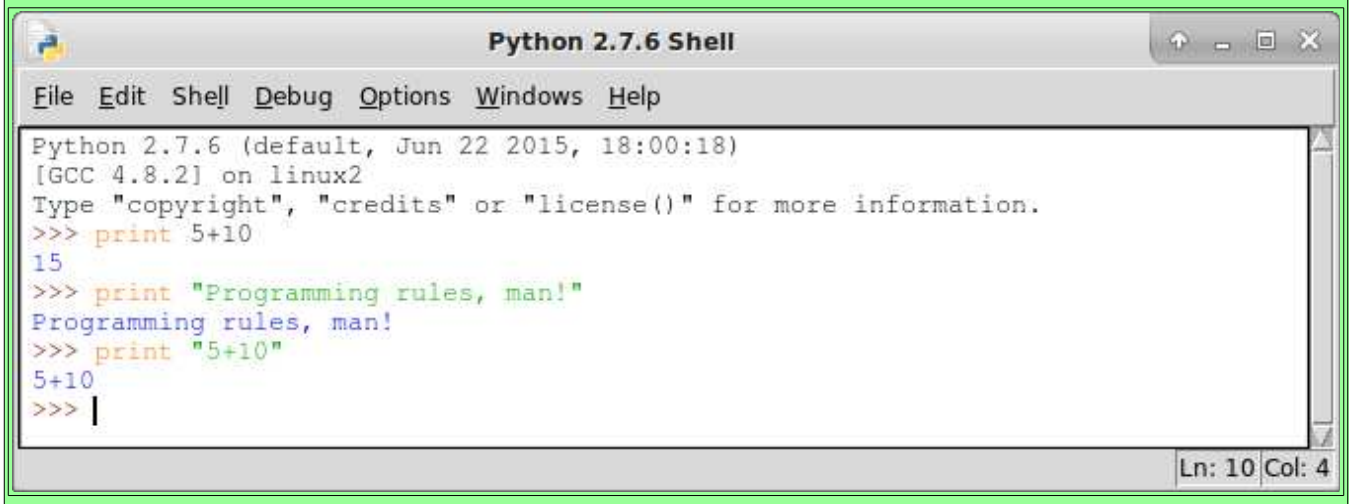
There is one more arithmetic operator in Python: the % operator. This operator returns the remainder of a division (similar to the **mod** operator in Scratch). Take a look at the following example:



The expression 56 / 10 is indeed 5 (the / operator produces an integer since both of the operands, 56 and 10, are integers). The remainder that results from the expression 56 / 10 is 6 (since 56 / 10 = 5 remainder of 6). Therefore, 56 % 10 = 6.

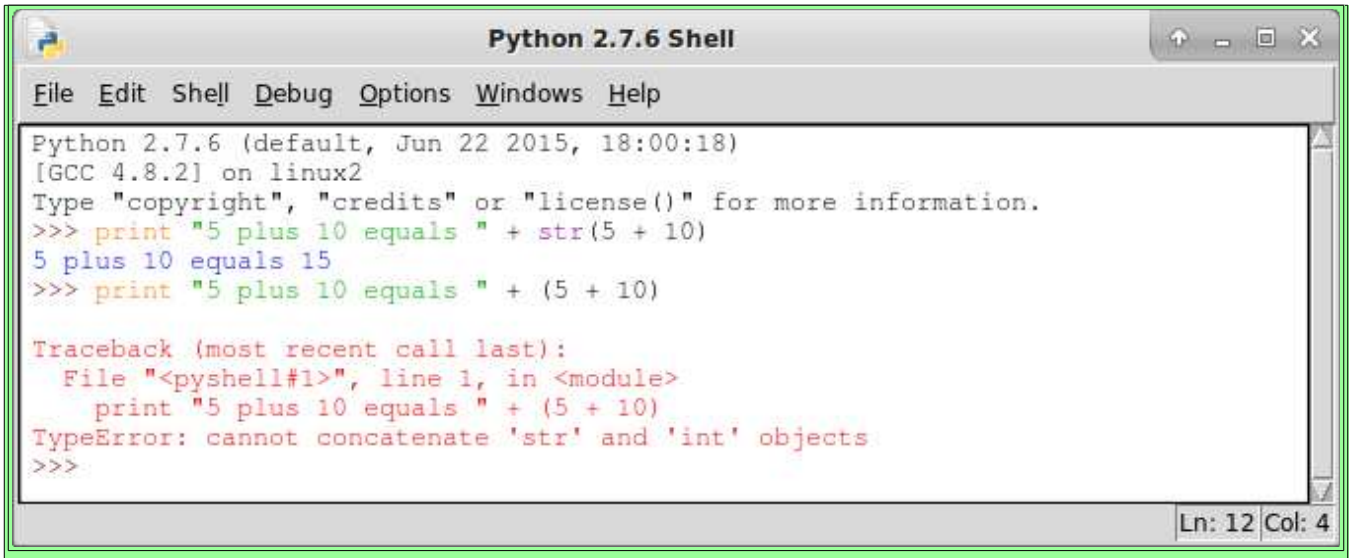
Output

As seen earlier, Python allows output via a **print** statement. Here are some simple examples:



The statement `print 5 + 10` instructs Python to display the result of the expression 5 + 10 (which is 15). The statement `print "Programming rules, man!"` does just what it did when shown earlier. Take a look at the last statement: `print "5+10"`. It looks suspiciously like the first statement, except that the expression 5 + 10 is enclosed in quotes. This lets the Python interpreter know that the characters "5 + 10" are to be interpreted as a string (characters strung together) as opposed to an arithmetic expression consisting of the + operator and the two operands, 5 and 10. This is why the output of this statement is, quite literally, 5 + 10.

Suppose that you would like to print the following string of characters: 5 plus 10 equals 15 (and that you would like for 15 to be calculated as the result of the expression 5 + 10). This can be accomplished as follows:

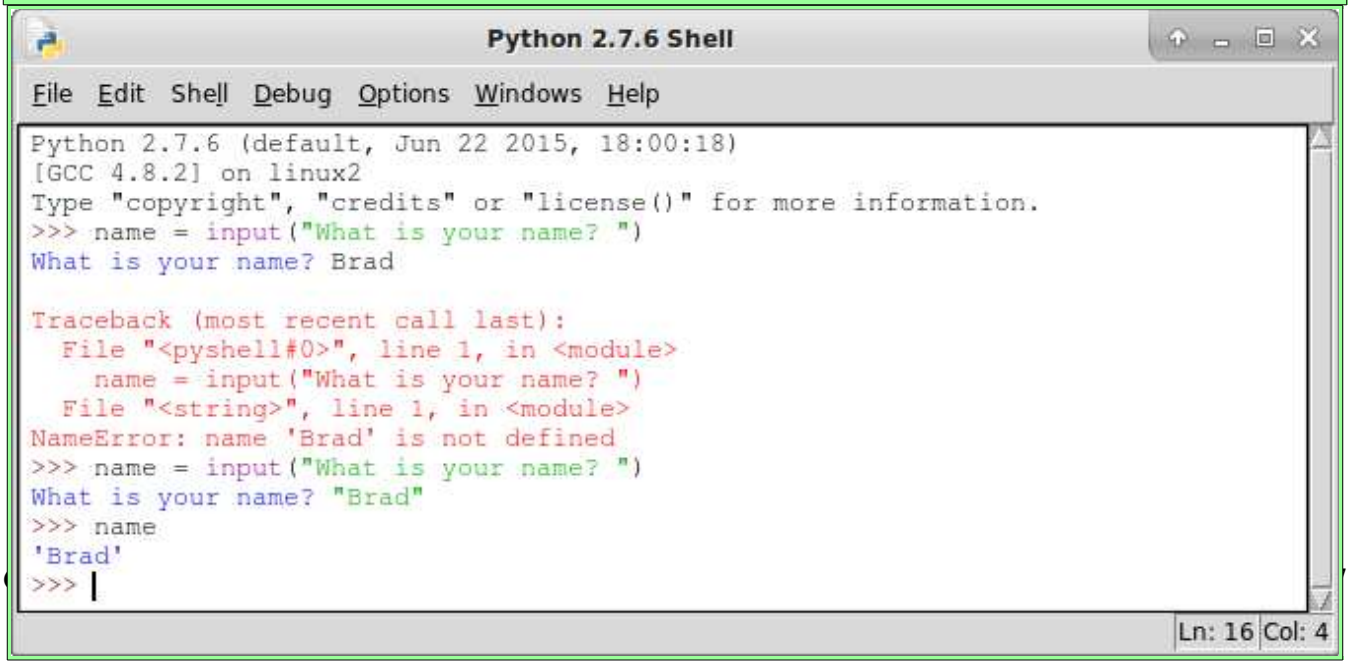


First, note the error produced by the following statement: `print "5 plus 10 equals " + (5 + 10)`. The error occurs because Python does not know how to "add" or concatenate the string "5 plus 10 equals " to the integer that results from the expression (5 + 10). To instruct the Python interpreter to concatenate the result of this expression (as characters) to the first part of the string, the result (15) must be converted to a string. In Python, this is accomplished with the **str()** function. Python converts anything within the parentheses (we call this the parameters of the function) to a string of characters. Therefore, the expression str(5 + 10) instructs the Python interpreter to first add 5 and 10 (to produce the result 15), and then convert 15 to the string "15". It is then valid to concatenate the string "5 plus 10 equals " to the string "15".

You may also have noticed that, for most of the examples, operands and operators were separated by a space. For example, the expression 5+10 was written in the Python shell as 5 + 10 (with spaces). This is an example of good coding style that increases the readability of our programs.

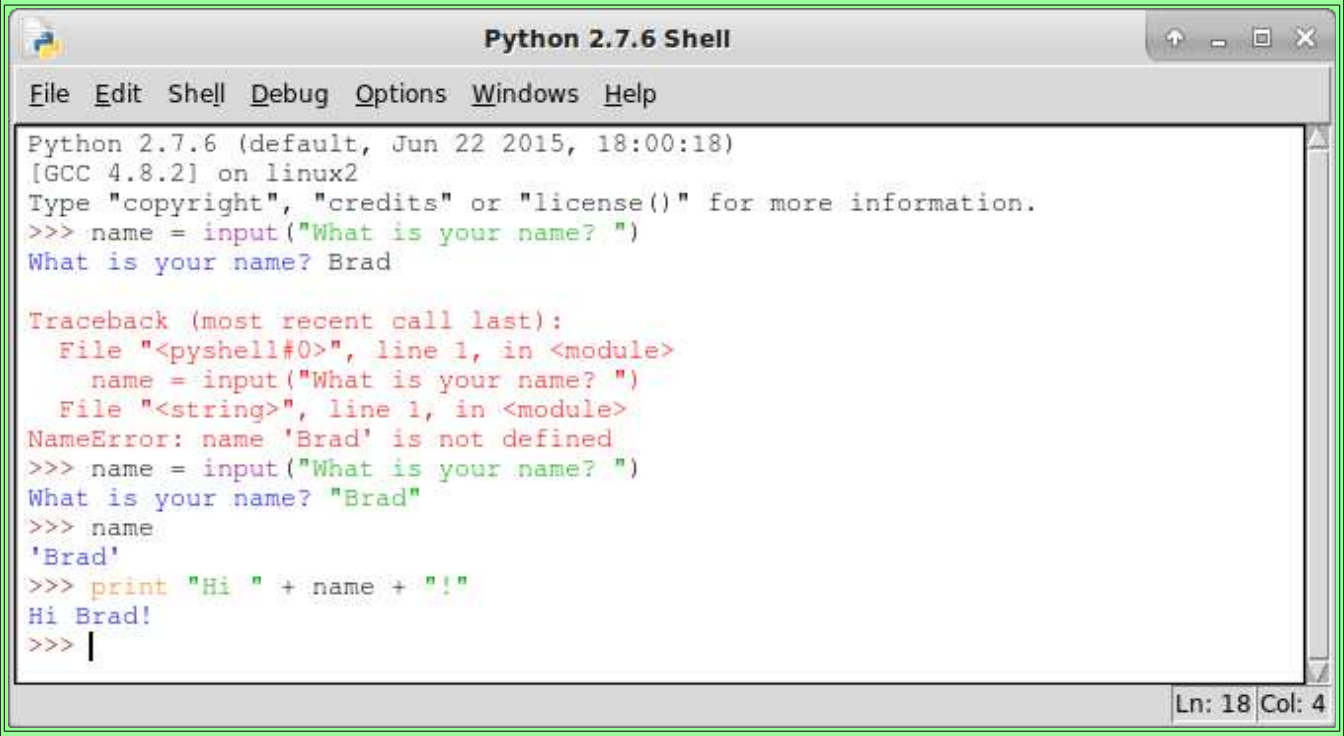
Input

Python also supports statements that allow users to input information via the **input()** function. This information is typically stored in variables. Take a look at the following example:

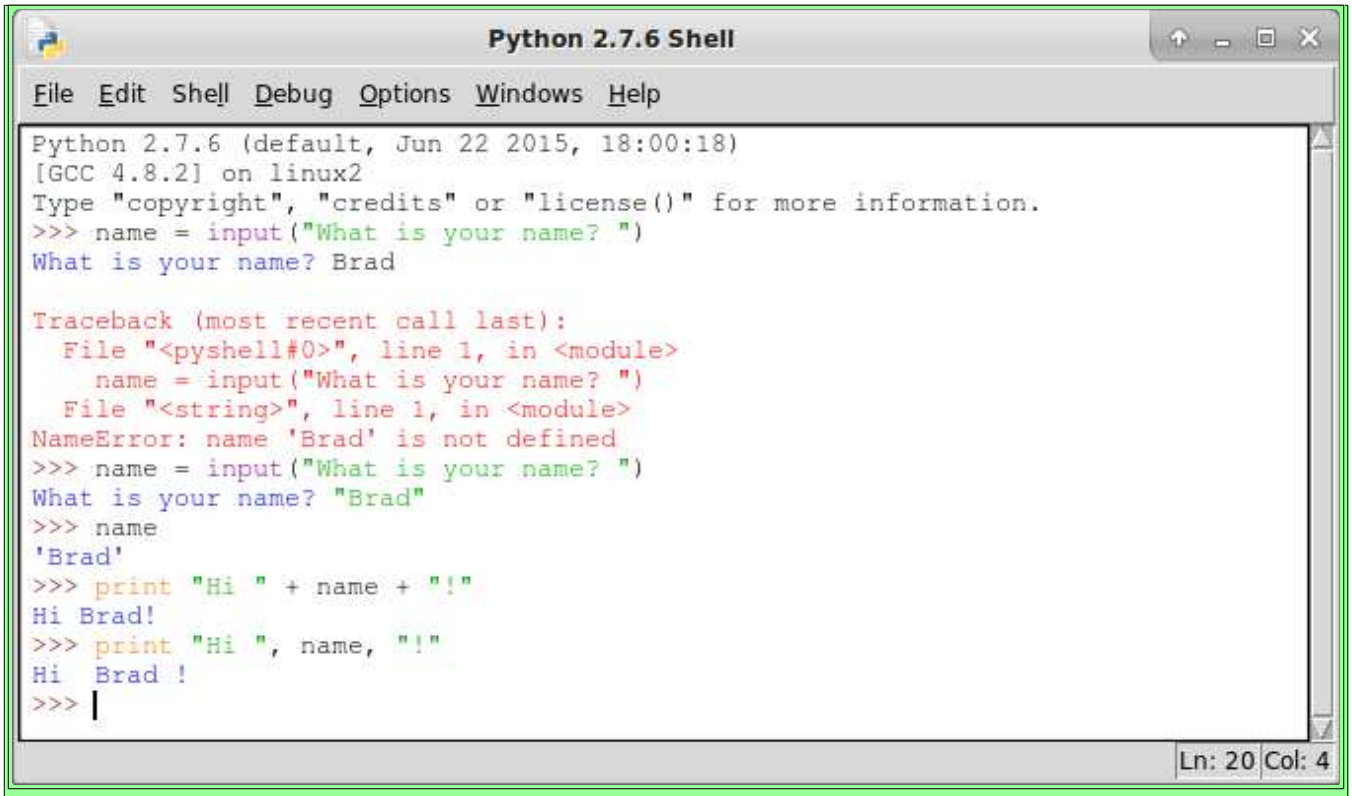


The statement, `name = input("What is your name? ")`, prompts the user for a name. It stores the result in the variable *name*. Note that it expects any type. We could have very well entered in 23 as the response, and the variable name would have stored the integer 23. But, of course, we wanted to enter an actual name. Since a name is a string of characters, then we must make sure to enclose the response with quotes. What happens if we forget to do so? Well, we get the error shown above. In the end, we can display the contents of the variable *name* simply by typing its name (i.e., *name*) in the Python shell. this is also shown above.

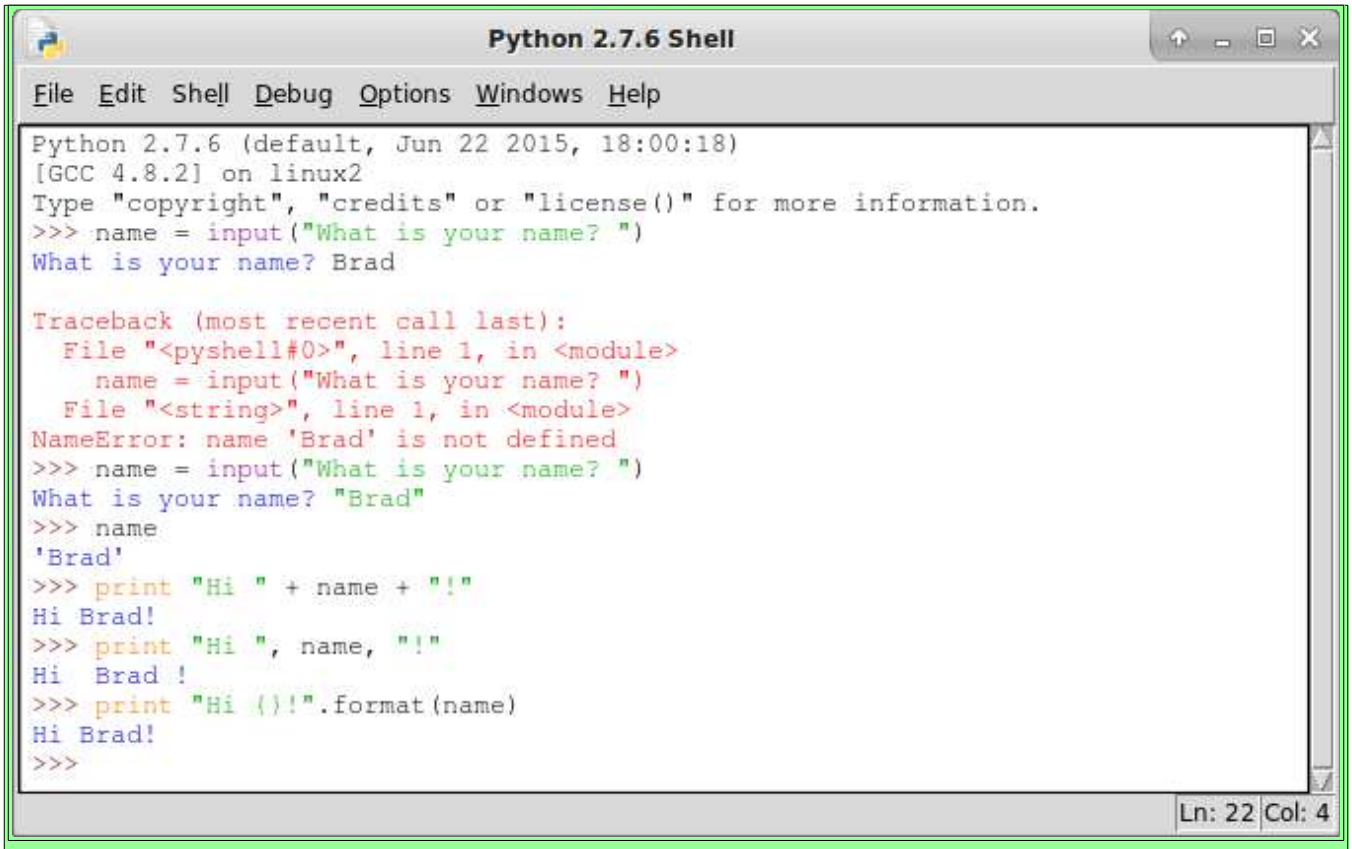
We can now use the variable *name* as follows:



That's another example of string concatenation. The **print** statement has another format:



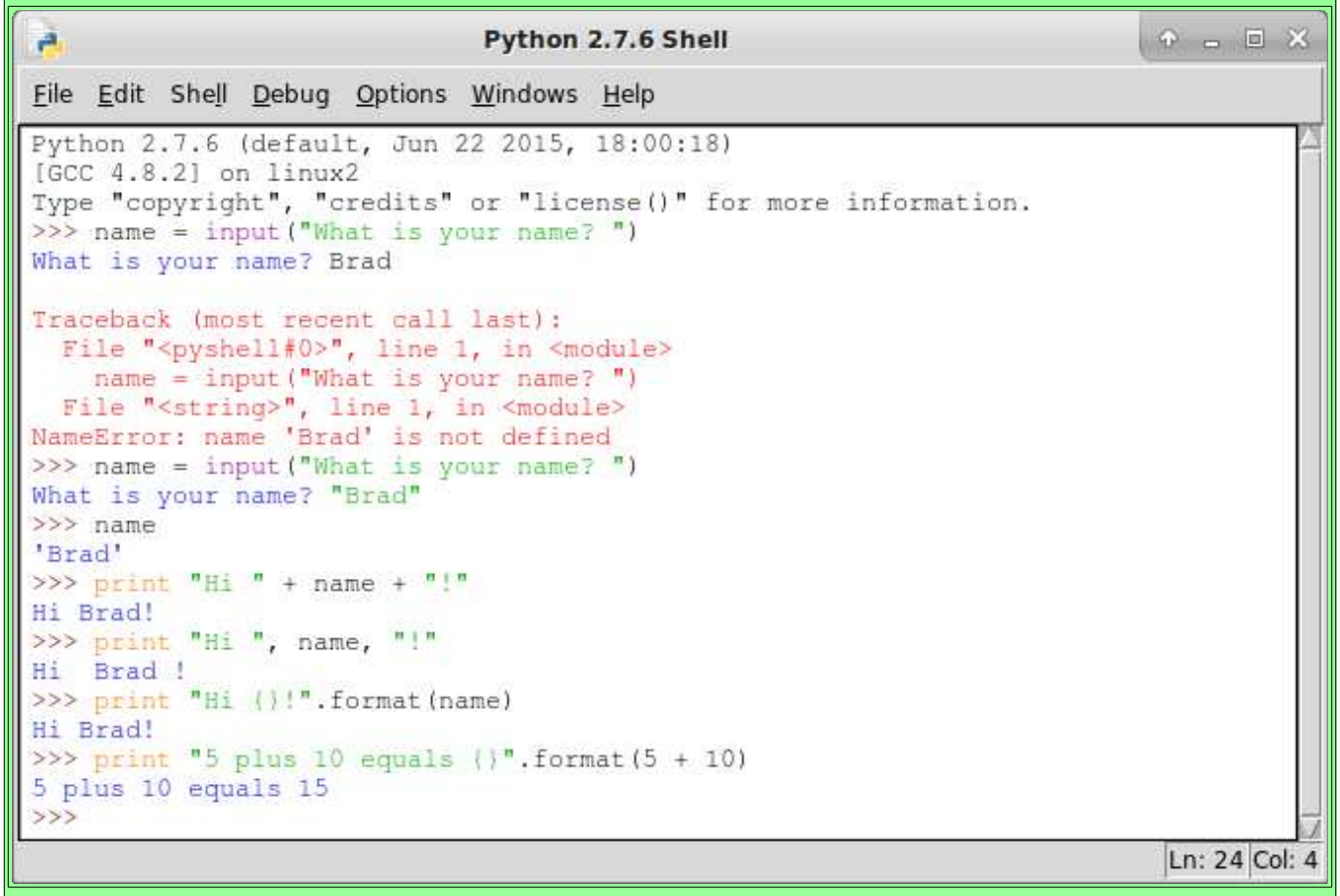
Note how we can separate the components of what we want output with commas. But note the difference! Apparently, Python inserts a space in between each component when the **print** statement is formatted in this manner. If this is not desired, we can modify the statement as follows:



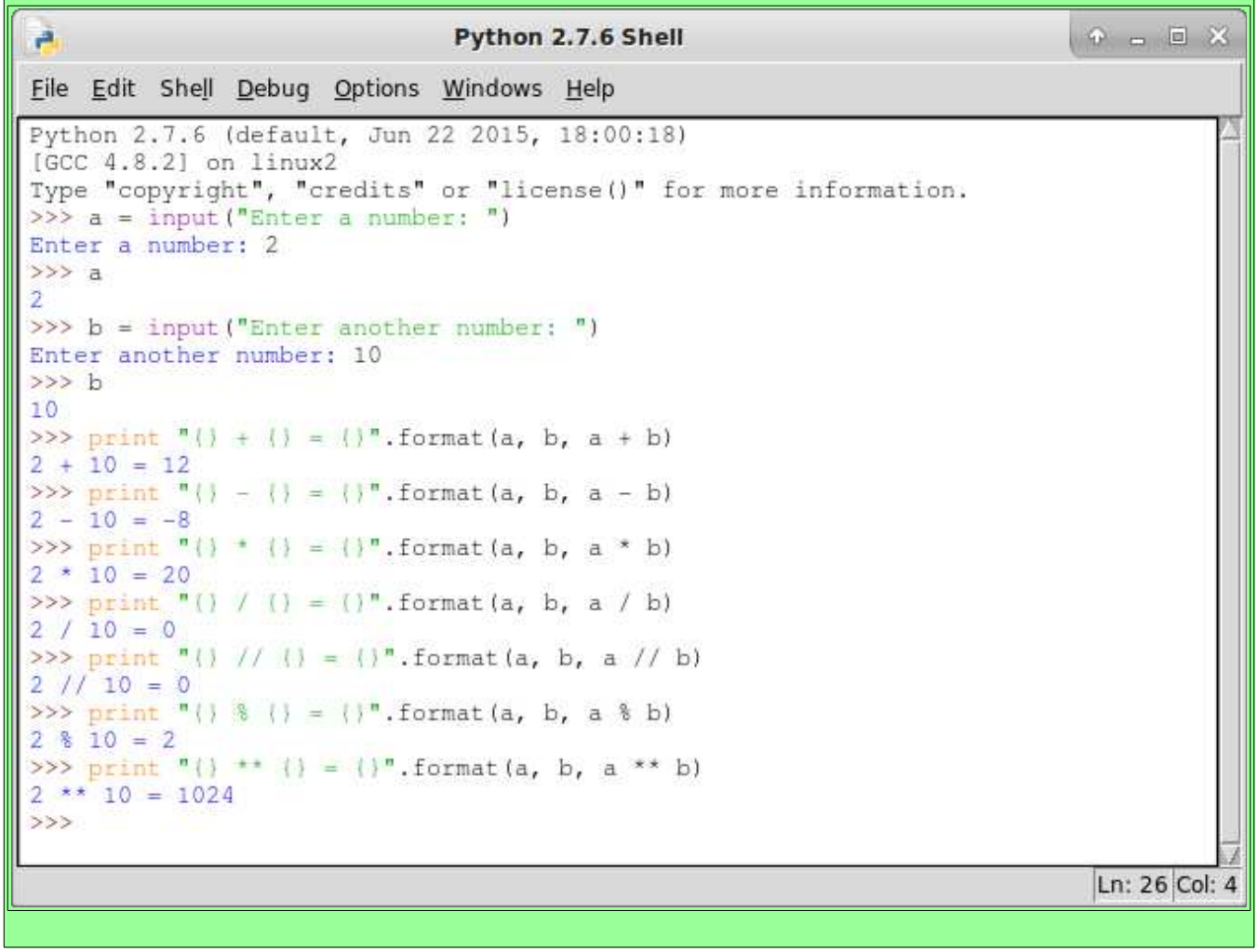
The braces ({})within a string are known as format fields. They are intended to note that something belongs there (that will be specified at a later time). To specify the contents to replace the braces with, we execute the **format** method on the string with the format field. We provide the values (which, in the example above, is just the contents of the variable *name*) that will be formatted to a string and replace the format fields. In the example above, the contents of the variable *name* is converted to a string (if necessary) and inserted in the string over the braces. This results in the output, "Hi Brad!"

Here's another example of this with the following statement:

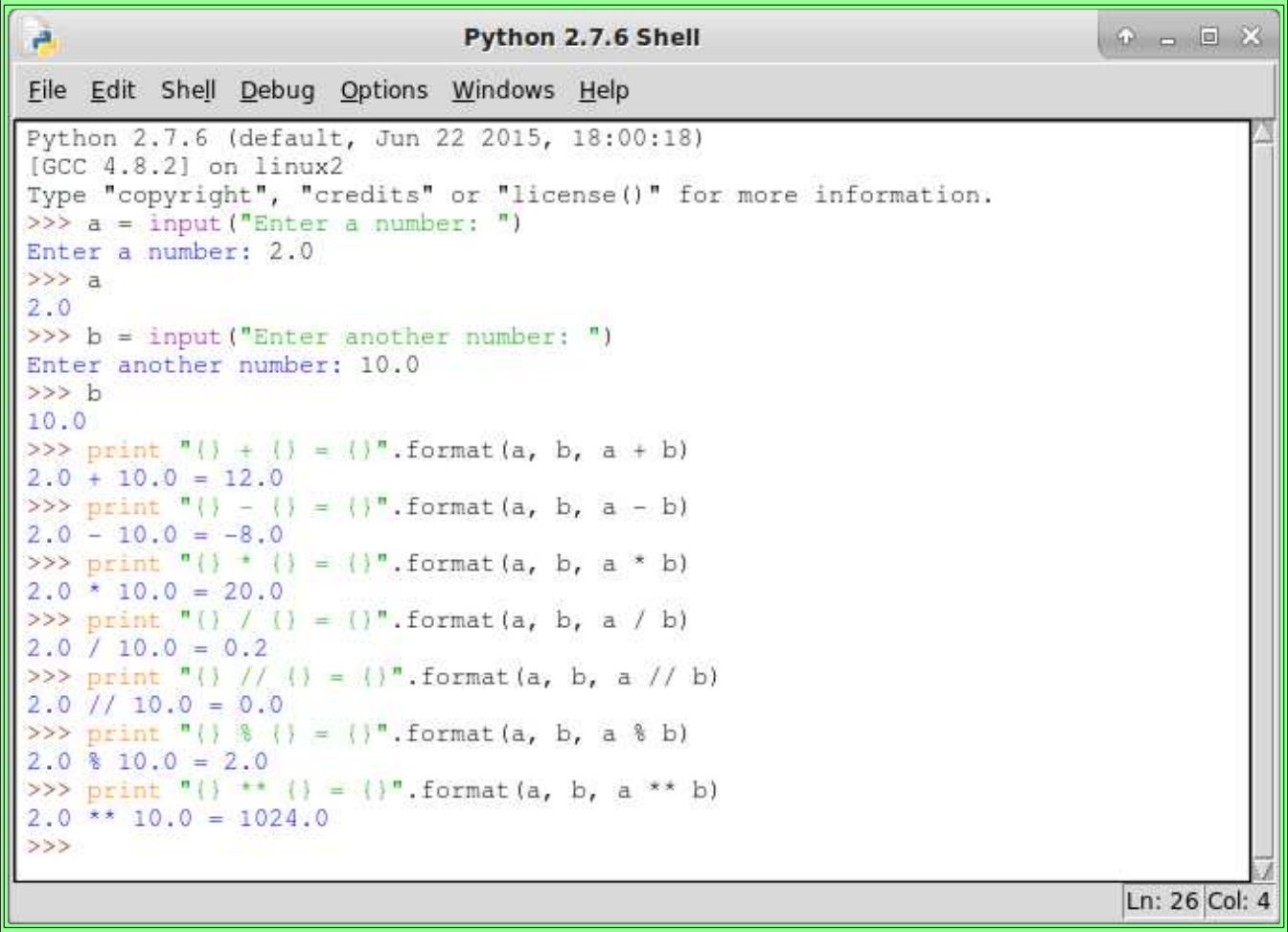
```
print "5 plus 10 equals {}".format(5 + 10):
```



Here are more examples of this with arithmetic expressions:



Try to do the same thing as in the example above, except set a=2.0 and b=10.0. Notice any differences?



Other than the variables and the results being expressed as decimals (i.e., floating point numbers), the statement, `print "{} / {} = {}".format(a, b, a / b)`, results in the exact result, 0.2, since at least one of the operands is a floating point value.

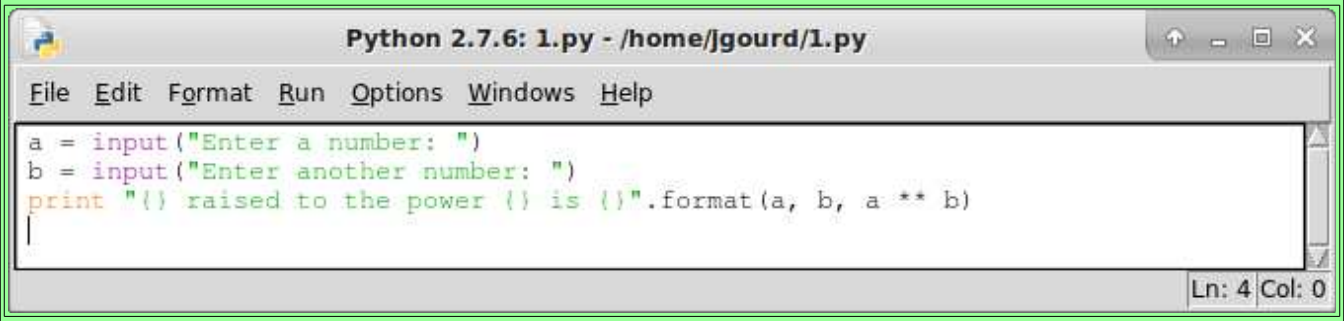
Creating programs and saving files

So far, we have been entering statements in the Python shell. These statements have been interpreted, one at a time. If we were to close the Python shell, everything that we entered would be lost. In order to save Python programs, we must type them in a separate editor outside of the Python shell, save them in a file. Once this has been done, we can then execute them in the Python shell.

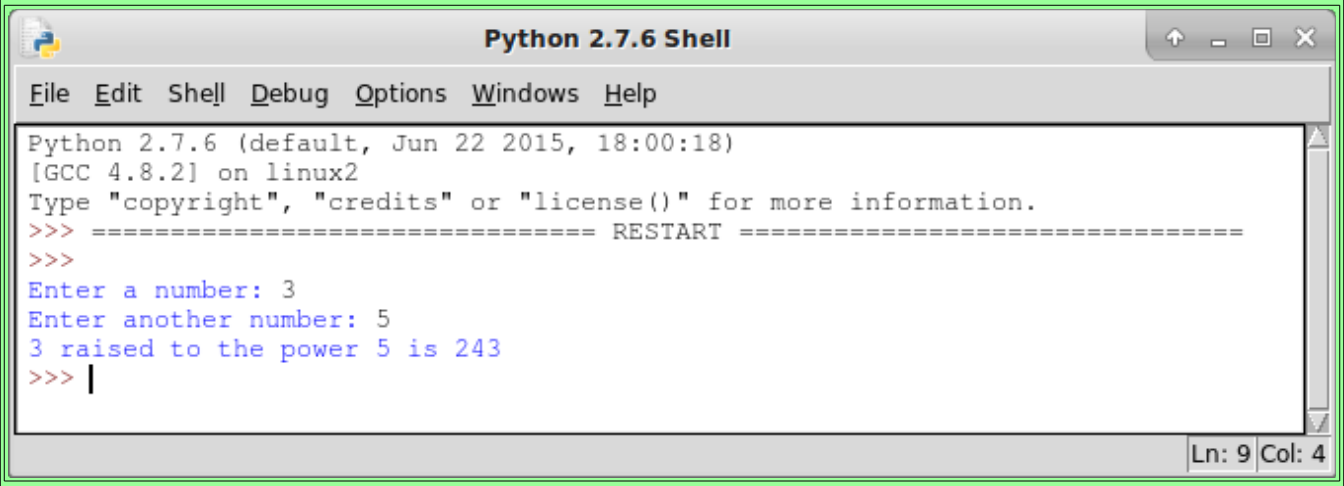
To create a new Python program, click on **File** | **New File** (or press **Ctrl+N**) in the Python shell. This brings up a new window (an editor that is a part of IDLE) in which we can type our program. Type the following program into this new window:

```
a = input("Enter a number: ")
b = input("Enter another number: ")
print "{} raised to the power {} is {}".format(a, b, a ** b)
```

This is what you should see at this point:



Before we can run this program, it must be saved. Do so by clicking on **File** | **Save** (or press **Ctrl+S**). Give it an appropriate name, and save it to an appropriate location. Now it can be executed by clicking on **Run** | **Run Module** (or by pressing **F5**). This executes the program in the Python shell:



Provide values for the two requested numbers (3 and 5 were provided in the example above).

Lastly, to exit IDLE, click on **File** | **Exit** (or press **Ctrl+Q**).

Reloading a saved file
To load a saved Python program, simply double-click on the saved file. This should bring up the IDLE editor with your file loaded in it. Sometimes, double-clicking on the file just opens it up in a notepad-like editor by default. To force it to open in IDLE, right-click the file instead, and select **Open with IDLE**. This should load it in the IDLE editor. The program can then be executed as before, by clicking on **Run** | **Run Module** (or by pressing **F5**). This will automatically open a Python shell and execute the program.

Your turn
Write a Python program that prompts the user for two numbers (let's use 14 and 3 for this example) and subsequently displays the following string:

```
The quotient of 14 divided by 3 is 4 with a remainder of 2.
```

Data types, constants, and variables
The kinds of values that can be expressed in a programming language are known as its **data types**. Recall that Scratch supports only two data types: text and numbers. Since Python is a general purpose programming language, it supports many more data types. Actually, it can support virtually any type that you can think of! That is, Python allows you to define your own type for use in whatever way you wish. Since this is user-defined, let's focus on what are called primitive types for now. The **primitive types** of a programming language are those data types that are built-in (or standard) to the language and typically considered as basic building blocks (i.e., more complex types can be created from these primitive types).

Python's standard types can be grouped into several classes: numeric types, sequences, sets, and mappings. Although there are actually others, we will focus on these in the Living *with* Cyber curriculum.

Numeric types include whole numbers, floating point numbers, and complex numbers. Python has two whole number types: `int` and `long`. The `int` data type is a 64-bit integer. The `long` data type is an integer of unlimited length. Note that in Python 3.x, an `int` is an integer of unlimited length (there is no `long` data type). These integer types can represent negative or positive whole numbers. The `float` type is a 64-bit floating point (decimal) number. Lastly, the `complex` type represents complex numbers (i.e., numbers with real and imaginary parts). Most of our programs will require only `int` and `float`.

So what does this all mean? We create variables that contain data of some data type. Knowing the data type of a variable is like knowing the superpowers of a person you can control. In this analogy, the superpowers of a data type are the methods and properties that can be leveraged for use in whatever program you are writing at the time. For example, one of the superpowers of the numeric data types is raising them to a power. To do that, we can use the function of the form `pow(x,y)`. In this example, *x* and *y* are variables that are of type `int` or `float`. The **pow** function returns the value of the computation involving raising the value in *x* to the power of the value in *y* (i.e., x^y). This function would not typically be able to work for variables that aren't numeric data types. You may recall that the same functionality can be implemented in Python as: `x ** y`. This effectively performs the same thing.

A **constant** is defined as a value of a particular type that does not change over time. In Python (just as in Scratch), both numbers and text may be expressed as constants. **Numeric constants** are composed of the digits 0 through 9 and, optionally, a negative sign (for negative numbers), and a decimal point (for floating point numbers). For example, the number -3.14159 is a numeric constant in Python.

A **text constant** consists of a sequence of characters (also known as a string of characters – or just a **string**). The following are examples of valid string constants:

"A man, a plan, a canal, Panama."
"Was it Eliot's toilet I saw?"
"There are 10 kinds of people in this world. Those who know binary, those who don't, and those who didn't know it was in base 3!"

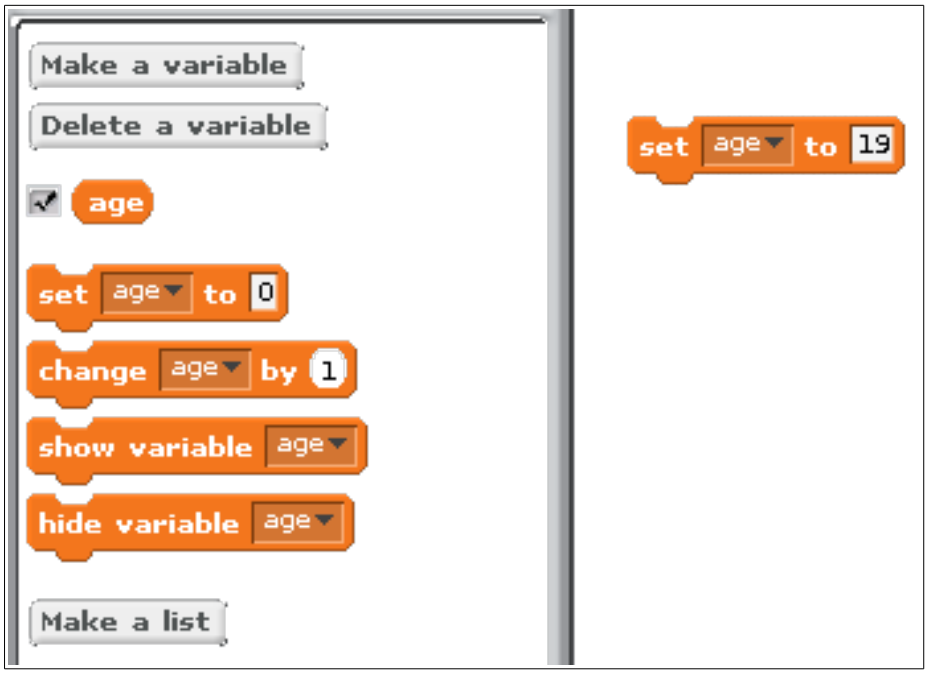
Note that, unlike Scratch, Python requires the quotes surrounding text constants.

A **variable** is defined to be a named object that can store a value of a particular type. Recall that Scratch supports two types of variables: text variables and numeric variables. Moreover, before a variable can be used, its name must be declared. In many programming languages, both its name and type must be

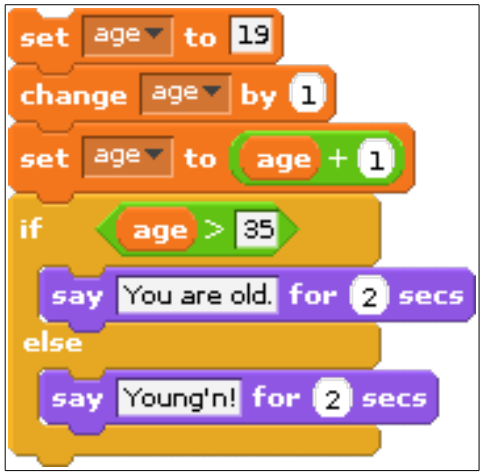
declared; however, both Scratch and Python only require a variable's name to be declared before it is used. Here is an example of declaring and initializing a variable in Python:

```
age = 19
```

In Scratch, the variable had to first be declared in the *variables* blocks group. A **set var to n** block was then used to initialize the variable:

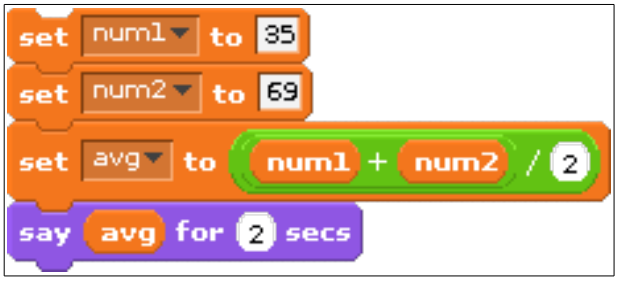


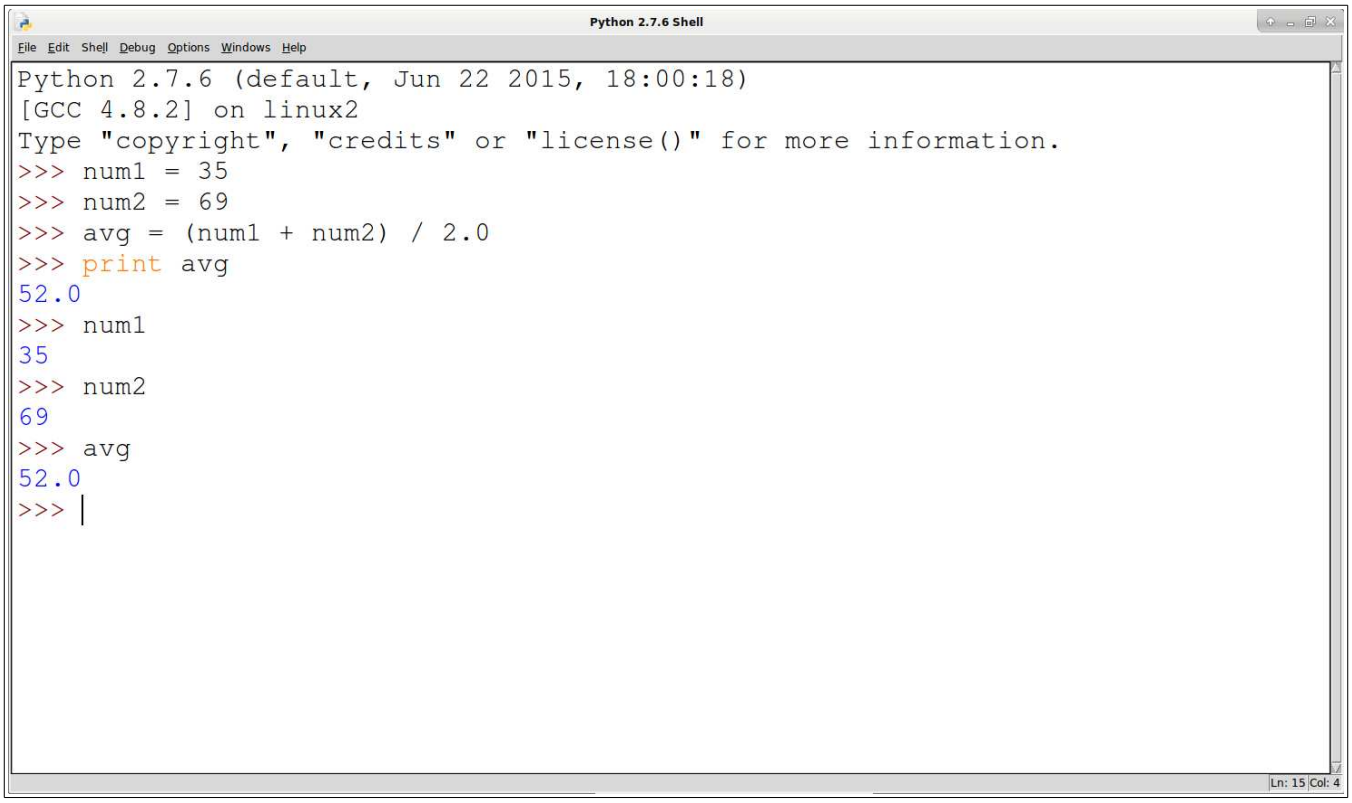
Here are some examples that deal with variables and how they compare in Scratch and Python:



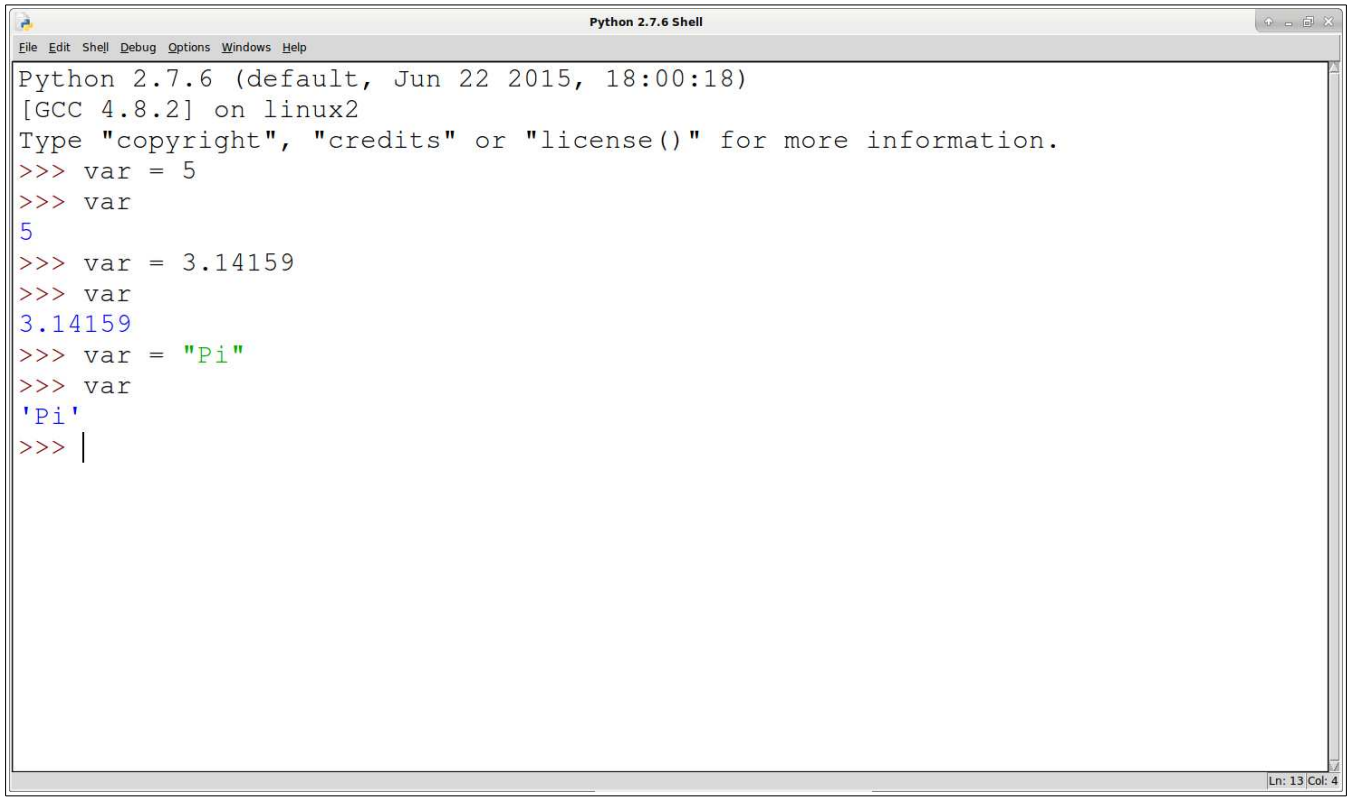
```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> age = 19
>>> age = age + 1
>>> age = age + 1
>>> if age > 35:
        print "You are old."
else:
        print "Young'n!"


Young'n!
>>> age
21
>>>
                                                                Ln: 16 Col: 4
```

Another example:





In short, to declare variables in Python, we simply write a statement that assigns a value to a variable name. Note that, just as in Scratch, we can assign a value of a different type to a variable. For example:



It is important to realize that, while human programmers generally try to give variables names that reflect the use to which they will be put, the variable name itself doesn't mean anything to the computer. For example, the numeric variable `age` can be used to hold any number, not just an age. It is perfectly legal for `age` to hold the number of students in a class or the number of eggs in your refrigerator. The computer couldn't care less. Human programmers, on the other hand, generally care a great deal. They expect a variable's name to accurately reflect its purpose; so while it is possible to do so, it would be considered poor programming practice to use the variable `age` to store anything other than an age.

Input and output

In order for a computer program to perform any useful work, it must be able to communicate with the outside world. The process of communicating with the outside world is known as input/output (or I/O). Most imperative languages include mechanisms for performing other kinds of I/O such as detecting where the mouse is pointing and accessing the contents of a disk drive.

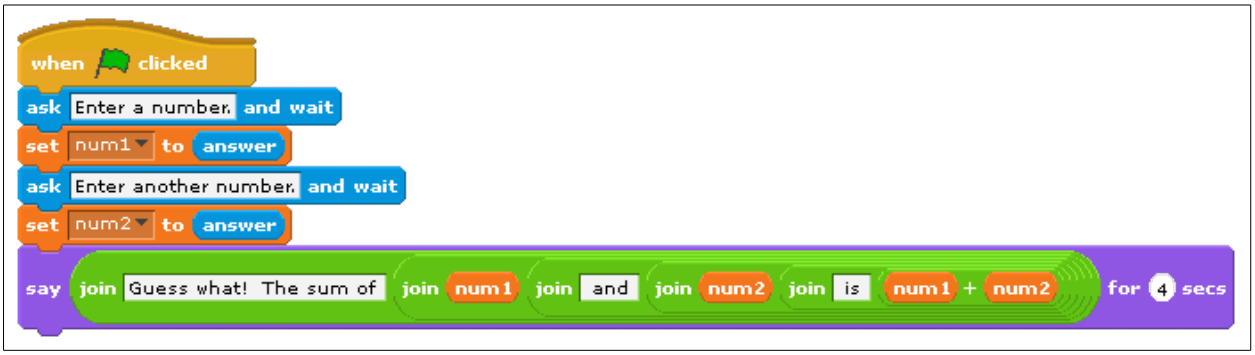
The flexibility and power that input statements give programming languages cannot be overstated. Without them the only way to get a program to change its output would be to modify the program code itself, which is something that a typical user cannot be expected to do.

General-purpose programming languages allow human programmers to construct programs that do amazing things. When attempting to understand what a program does, however, it is vitally important to always keep in mind that the computer does not comprehend the meaning of the character strings it manipulates or the significance of the calculations it performs. Take, for example, the following simple Scratch program:

This program simply displays strings of characters, stores user input, and echoes that input back to the screen along with some additional character strings. The computer has no clue what the text string "Please enter your name: " means. For all it cares, the string could have been "My hovercraft is full of eels." or "qwerty uiop asdf ghjkl;" (or any other text string for that matter). Its only concern is to copy the characters of the text string onto the display screen.

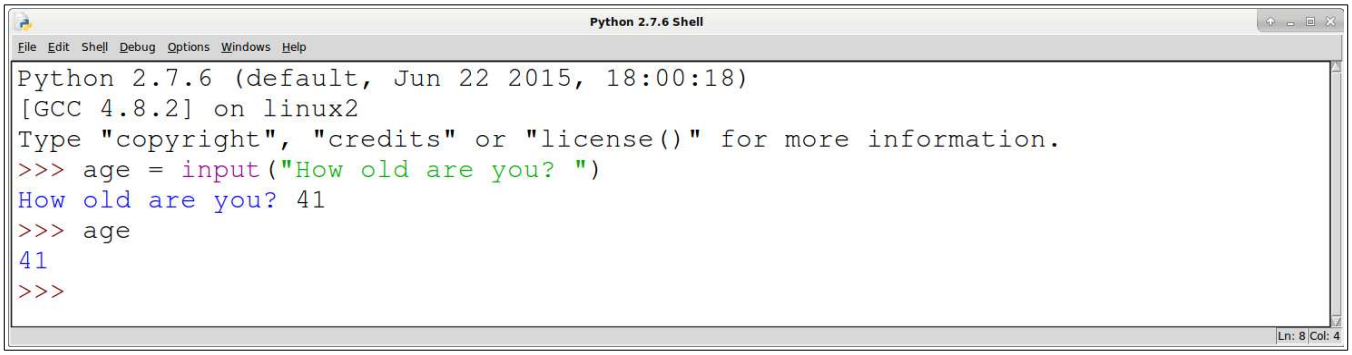
Only in the minds of human beings do the sequence of characters "Please enter your name: " take on meaning. If this seems odd, try to remember that comprehension does not even occur in the minds of all humans, only those who are capable of reading and understanding written English. A four year old, for example, would not know how to respond to this prompt because he or she would be unable to read it. This is so despite the fact that if you were to ask the child his or her name, he or she could immediately respond and perhaps even type it out on the keyboard for you.

Now consider this Scratch program:



Here, the input is numeric instead of text. The program prompts the user for two numbers, which it then computes the sum for and displays to the user. Note that two variables were declared: `num1` and `num2`. The first number is captured and stored in the variable `num1`. The second number is captured and stored in the variable `num2`. What do you think would happen if the user did not provide numeric input and, for example, inputted "Bob" for the first number? In the *real world*, programmers must create robust programs that examine user input in order to verify that it is of the proper type before processing that input. If the input is found to be in error, the program must take appropriate corrective action, such as rejecting the invalid input and requesting the user try again.

In Python, output is implemented as a **print** statement: `print "This is some output!"` We use the **input** statement to ask a question and obtain user input. In the same statement, we can assign the result of this to a variable:



Of course, we need to take care to properly specify whether the input is numeric or text (i.e., with quotes).

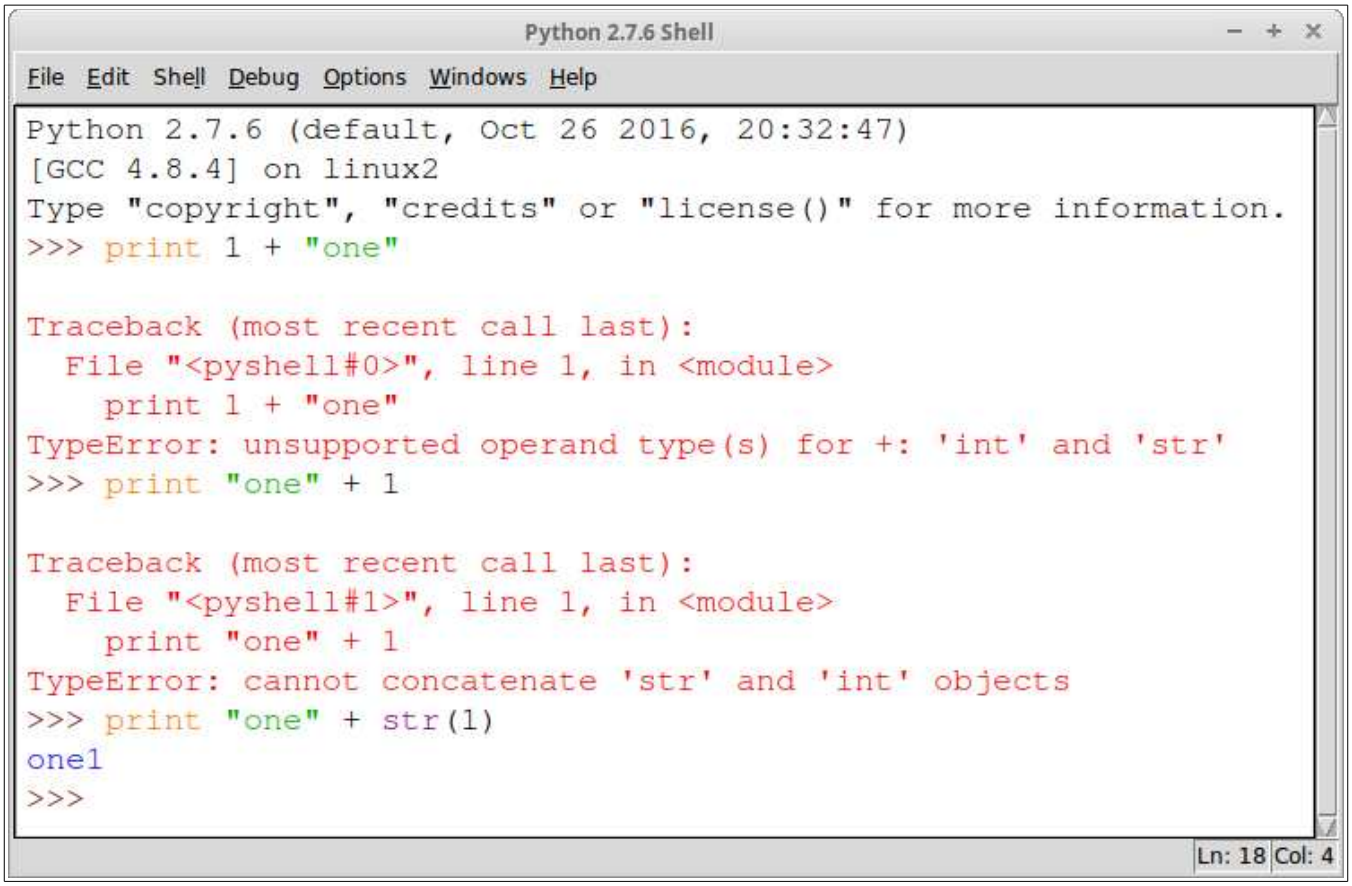
Expressions and assignment

You've seen how to assign values to variables above using a simple assignment statement. For example:

```
name = "Shonda Lear"
age = 19
grade = 91.76
letter_grade = "A"
```

These are all examples of assignment statements. In this configuration, the equal sign (=) functions as the assignment operator. Later, you will see how it can also be used to compare values or expressions.

An **expression** in a programming language is some combination of values (e.g., constants and variables) that are evaluated to produce some new value. For example, a simple expression in Python is `1 + 2`. The result of this expression is, of course, 3! Expressions usually take on the form of *operand operator operand*. In the previous example, the operator was + and the operands were 1 and 2. The operator + has a very well defined behavior on operands of numeric types: it simply adds them. On string types, it concatenates. What do you think would happen if the operands are of two different types (e.g., numeric and string)? Let's see:



As expected, Python doesn't know what it means to "add" a numeric type to a string type. Therefore, it results in an error: unsupported operand type(s). To "add" a string type to a numeric type, we must convert the numeric type to a string type via `str`. Then, Python understands that "adding" actually means concatenating two strings. It is interesting to note how Python handles "multiplying" a string type and a numeric type like this:

```
print "hello" * 5
```

What do you suppose the output will be? Would you expect an error? Or would you expect something like this:

```
hellohellohellohellohello
```

It turns out that Python understands how to "multiply" both types by interpreting the `*` operator as concatenating a string type a number of times specified by a numeric type. Therefore, the statement `print 10 * "hello"` concatenates the string type "hello" 10 times!

Python has many different operators that perform a variety of operations on operands. These will be discussed later in this lesson.

Subprograms

A **subprogram** is a block or segment of organized, reusable, and related statements that perform some action. It is essentially a program within a program. Recall an earlier lesson on representing algorithms as to-do lists. One algorithm represented the steps necessary to *get to class*. One of those steps was *eat breakfast*. We noted how we could zoom in to that step and identify the sub-steps necessary to complete the *eat breakfast* step. Control flow shifted from the main to-do list to the *eat breakfast* to-do list when the *eat breakfast* step was encountered, and then returned to the main to-do list at the point where it left earlier. We can consider the *eat breakfast* to-do list as a subprogram.

Very few *real* programs are written as one long piece of code. Instead, traditional imperative programs generally consist of large numbers of relatively simple subprograms that work together to accomplish some complex task. While it is theoretically possible to write large programs without the use of subprograms, as a practical matter any significant program must be decomposed into manageable pieces if humans are to write and maintain it.

Subprograms make the construction of software libraries possible. A **software library** is a collection of subprograms, or *routines* as they are sometimes called, for solving common problems that have been written, tested, and debugged. Most programming languages come with extensive libraries for performing mathematical and text string operations and for building graphical user interfaces. These languages allow programmers to include library routines in their code. Using subprograms from the library speeds up the software development process and results in a more reliable finished product.

When a subprogram is invoked, or called, from within a program, the *calling* program pauses temporarily so that the *called* subprogram can carry out its actions. Eventually, the called subprogram will complete its task and control will once again return to the *caller*. When this occurs, the calling program *wakes up* and resumes its execution from the point it was at when the call took place.

Subprograms can call other subprograms (including copies of themselves as we will see later). These subprograms can, in turn, call other subprograms. This chain of subprogram invocations can extend to an arbitrary depth as long as the *bottom* of the chain is eventually reached. It is necessary that infinite calling sequences be avoided, since each subprogram in the chain of subprogram invocations must eventually complete its task and return control to the program that called it.

Subprograms are broken down into two types: methods and functions. Generally, a **method** is a subprogram that performs an action and returns flow of control to the point at which it was called. A **function** is similar; however, it returns some sort of value before flow of control is transferred back to the point at which it was called. For example, a method may simply display some useful information about a program to the user (e.g., a program's help menu), while a function may compute some numeric value and return it to the user. Subprograms in Python are generally just referred to as functions, regardless of whether or not they return a value. For the remainder of this lesson, we will refer to subprograms as functions.

In Python, functions must formally be declared prior to their use. That is, the body or content of a function must be specified in a program before it can be called. The syntax for declaring a function is as follows:

```
def function_name(optional_parameters):
    function_body
```

The keyword **def** is a reserved word in Python and is used to declare functions. A function name can be any valid identifier. As you will see later, an **identifier** is a name used to identify a variable, function, or other object (note that objects will be discussed later). The function name must be followed by a set of parentheses containing optional parameters. **Parameters** allow for values (constants, variables, expressions, and so on) to be passed in to a function. For example, a function may accept two values, calculate their average, and return the result to the caller. The function definition is terminated with a colon (:). The body of a function (i.e., its enclosed statements) is indented.

Here is an example of a simple function that displays a line of text:

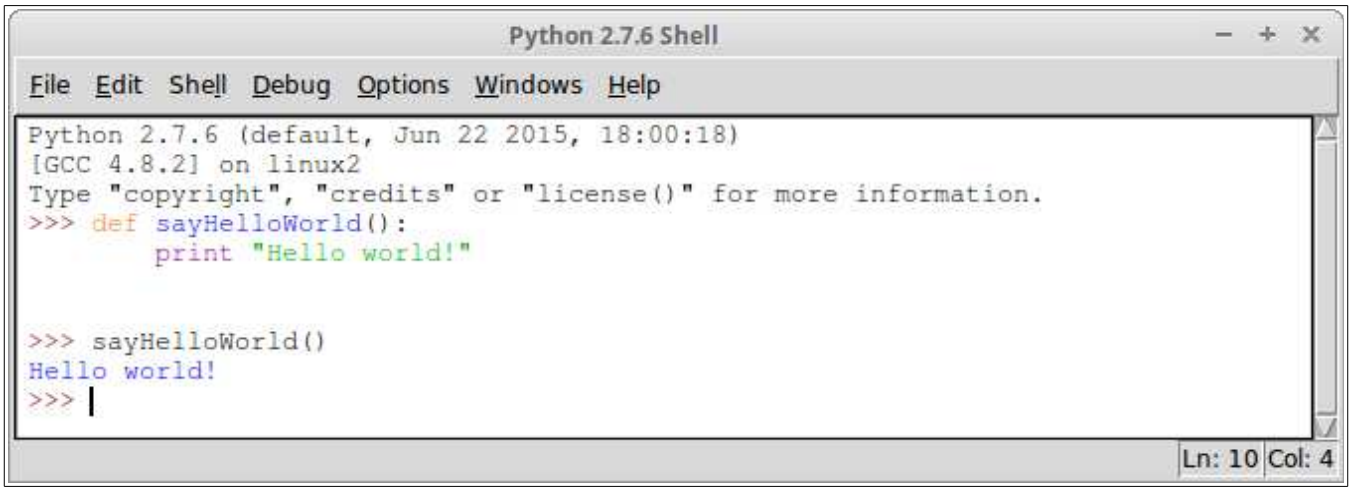
```
def sayHelloWorld():
    print "Hello world!"
```

This function is called `sayHelloWorld` and takes no parameters. It simply displays the text, "Hello world!"

To call this function, we simply need to specify its name and the values of its parameters (if any) as follows:

```
sayHelloWorld()
```

Here is sample output of calling this simple function:



Formally, functions have a **header** and a **body**. The header is the statement that defines the function (i.e., with the **def** keyword). The header of a function is often called its **signature**, and provides its name and any parameters. Function parameters help a function complete its task by providing input values. In fact, each call to a function possibly means a new set of parameters. Some functions compute and return a result, called the **return value**, that is returned via the **return** keyword.

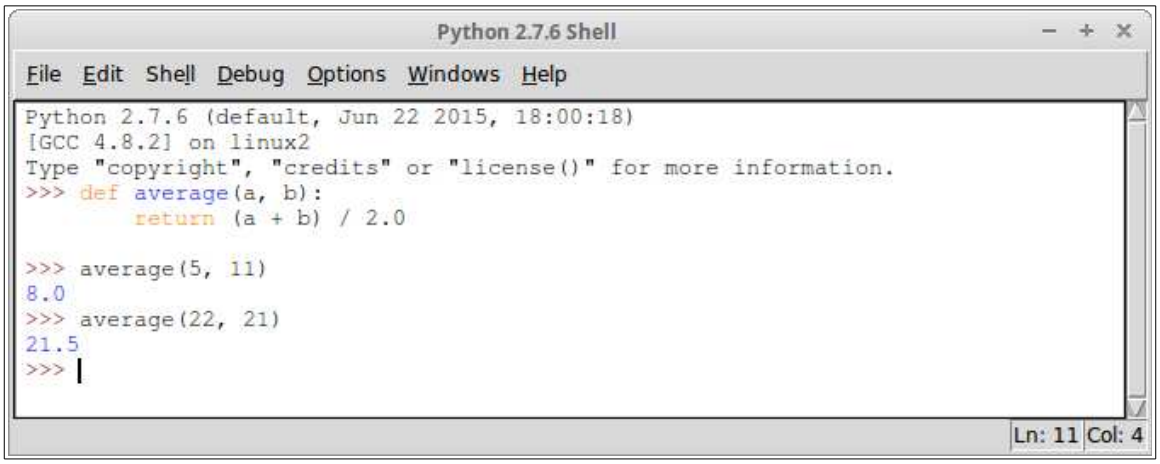
Here's a function that accepts two parameters and calculates (and returns) the average of the two:

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

And here's how it could be called:

```
average(5, 11)
```

The output of this (and another example) is shown below:



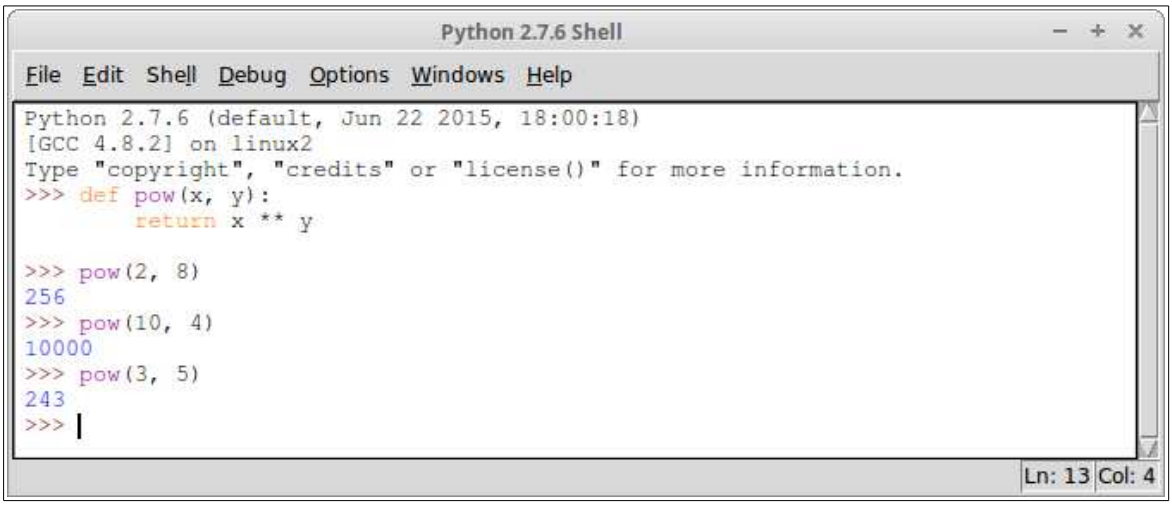
Note the **return** keyword. Its purpose is to return whatever expression comes after it. The statement `return (a + b) / 2.0` returns the result of the expression `(a + b) / 2.0` to the caller (which happened at the statement `average(5, 11)`).

Here's a `pow` function that returns the exponentiation of one parameter by another:

```
def pow(x, y):
    return x ** y
```

Recall that `x ** y` in Python implies exponentiation and means *x* raised to the power *y*. Formally, we call `**` the exponentiation operator. Operators will be discussed in detail in the next section.

Here's sample output of this function with various parameters:

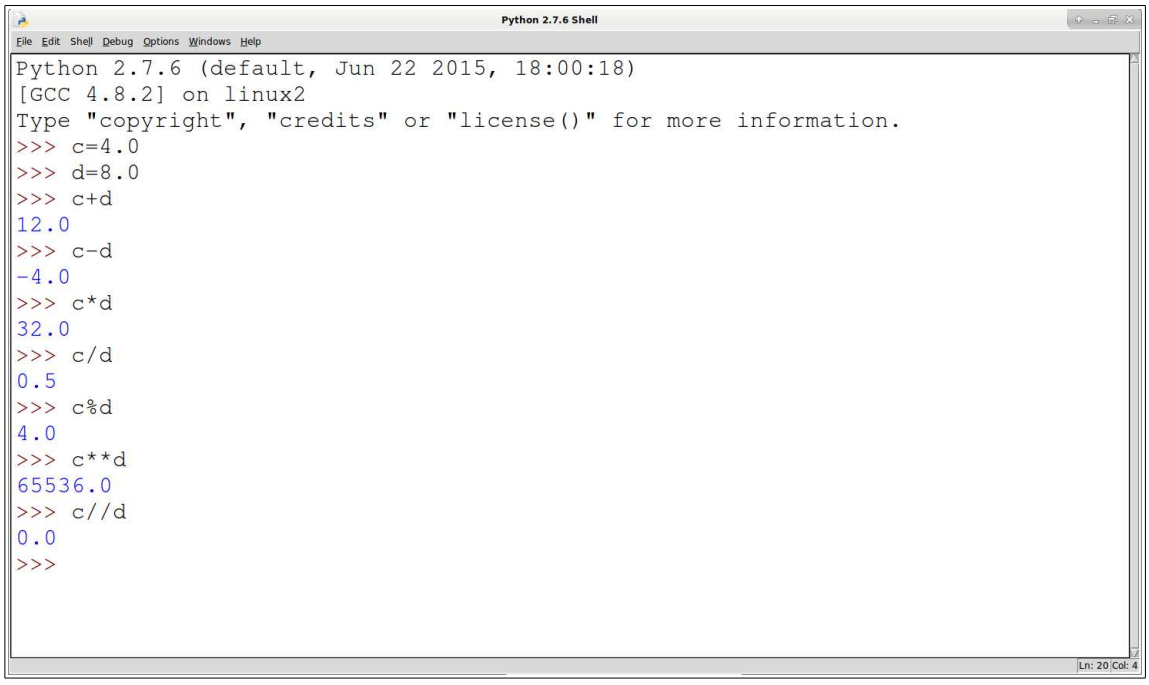


Operators

Like Scratch, Python has a variety of operators, broken down into several classes: arithmetic operators, relational (comparison) operators, assignment operators, logical operators, bitwise operators, membership operators, and identity operators. Operators allow operations to be performed on operands. Let's first take a look at the arithmetic operators since they relate directly to assignment. The **arithmetic operators** allow us to perform arithmetic operations on two operands. In the following table, assume that a = 23, b = 17, c = 4.0, and d = 8.0:

Python Arithmetic Operators and Examples			
+	addition	a + b = 40	c + d = 12.0
–	subtraction	a – b = 6	c – d = -4.0
*	multiplication	a * b = 391	c * d = 32.0
/	division	a / b = 1	c / d = 0.5
%	modulus	a % b = 6	c % d = 4.0
**	exponentiation	a ** b = 141050039560662968926103L	c ** d = 65536.0
//	floor division	a // b = 1	c // d = 0.0

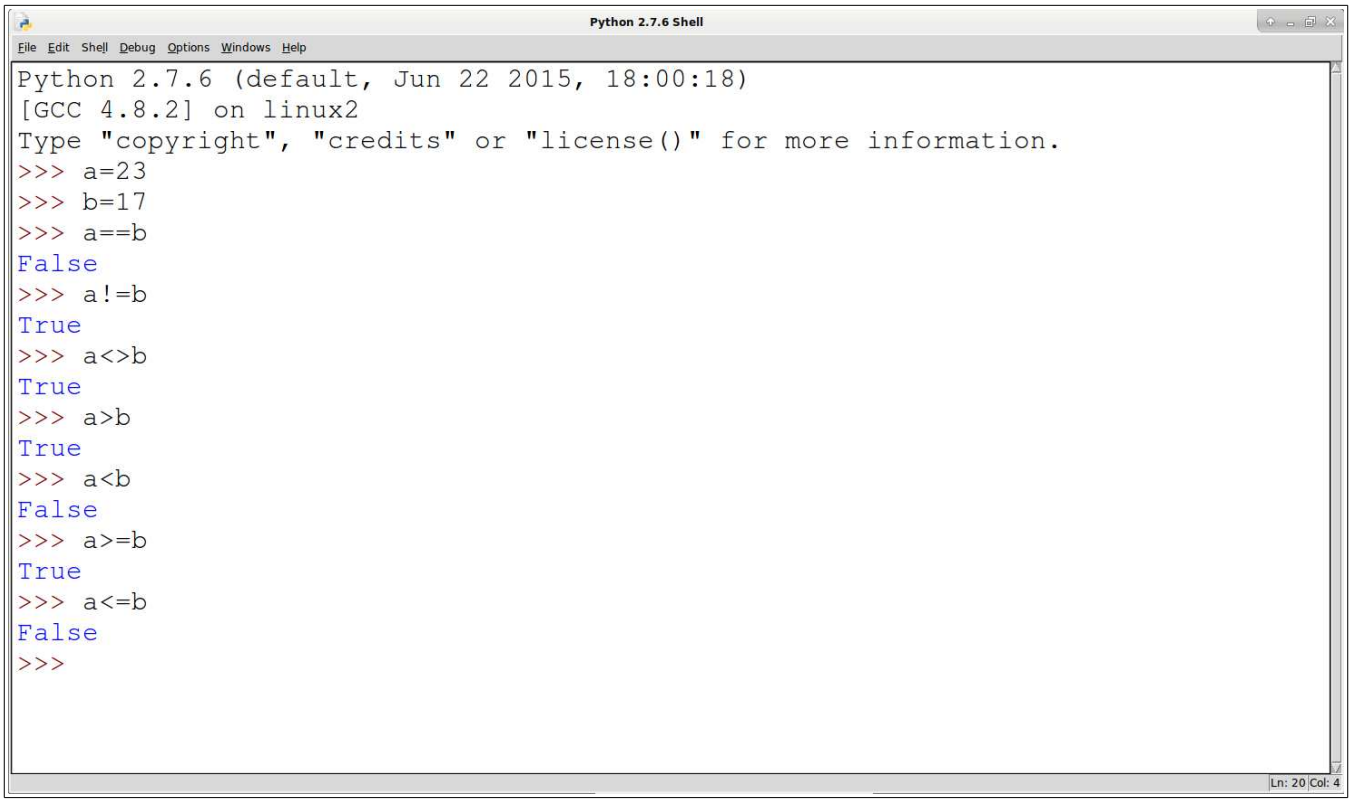
Note the L at the end of the result of the expression a ** b. In Python 2.x, 64-bit integers are of type `int`, and unlimited length integers are of type `long`. An L at the end of a number implies that it is of the `long` type. Here is output of the examples in the previous table using the variables c and d in IDLE:



The **relational operators** allow us to compare the values of two operands. The result is the relation among the operands. In the following table, assume that a = 23 and b = 17:

Python Relational Operators and Examples		
==	equality	a == b is False
!=	inequality	a != b is True
<>	inequality	a <> b is True
>	greater than	a > b is True
<	less than	a < b is False
>=	greater than or equal to	a >= b is True
<=	less than or equal to	a <= b is False

Note that the capitalization of `True` and `False` is intentional. In Python, the boolean value *true* is expressed as `True` and *false* as `False`. Here is output of the examples in the previous table in IDLE:

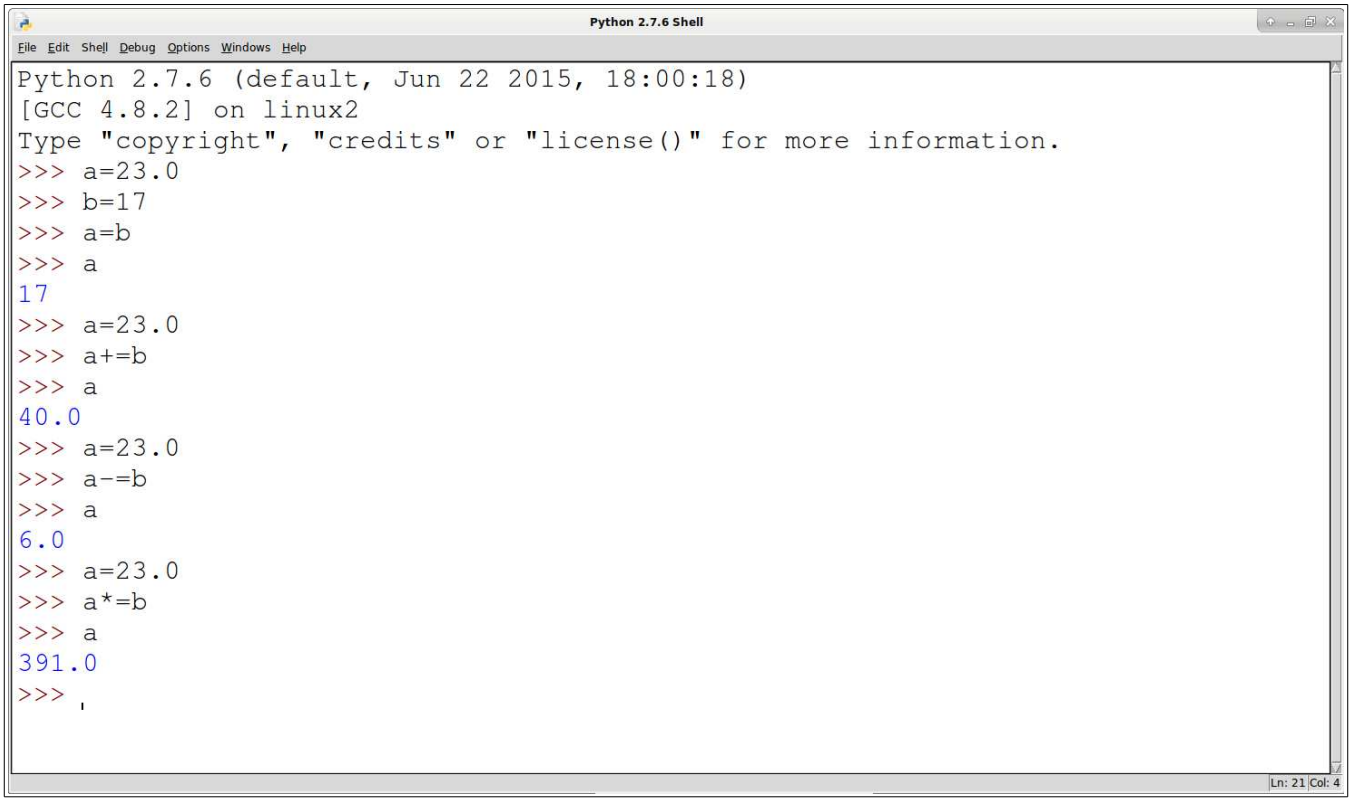


In Python, relational operators are typically used in if-statements, where branching is often desired. This will be illustrated in more detail later.

The **assignment operators** allow us to assign values to variables. You have already seen the most basic example of this using the equal assignment operator (as in the statement: `age = 19`). In the following table, assume that a = 23.0 and b = 17:

Python Assignment Operators and Examples		
=	a = b assigns b to a	a = 17
+=	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
**=	a **= b raises a to the power b and stores the result in a (same as a = a ** b)	a = 1.4105003956066297e+23
//=	a //= b divides a by b and stores the floor of the result to a (same as a = a // b)	a = 1.0

Here is output of the examples in the previous table in IDLE:



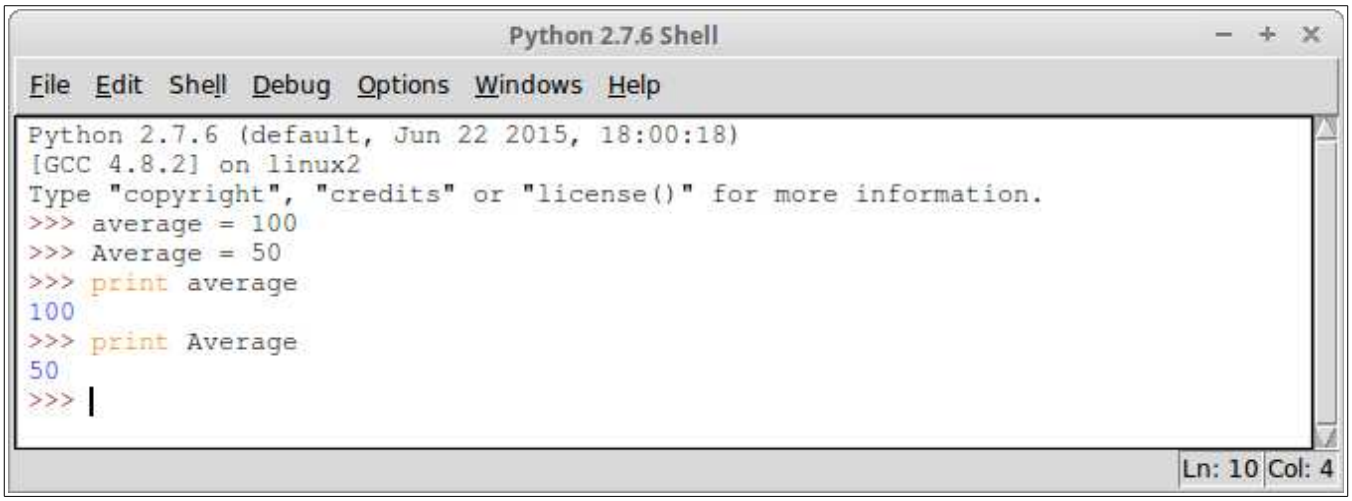
Identifiers and reserved words

An **identifier** is a name used to identify a variable, function, or other object (note that objects will be discussed later). Variable names (such as `age` and `average`, for example) or function names (such as `midPoint` and `distance`, for example) are all valid identifiers.

In Python, identifiers must begin with a letter (either lowercase *a* to *z* or uppercase *A* to *Z*) or an underscore (_) followed by zero or more letters, underscores, and digits (0 through 9). Here are examples of valid identifiers:

```
average
Average
average_grade
averageScore
_mustard
_7a69_
a1b2c3X7Y9Z0
```

Note that Python is a case-sensitive language. For example, the identifier `average` is not the same as the identifier `Average`. Take a look at this example:



Reserved words (sometimes called **keywords**) in a programming language are words that are meaningful to the language and cannot be used as identifiers. Most programming languages have quite a few reserved words. Python 2.7, for example, has the following reserved words:

```
and       as        assert    break     class
continue  def       del       elif      else
except    exec      finally   for       from
global    if        import    in        is
lambda    not       or        pass      print
raise     return    try       while     with
yield
```

You are already familiar with some of these: `and`, `def`, `not`, `or`, `print`, and `return`. Many of the Python reserved words will be discussed later.

Comments
It is often useful to provide informative text in our programs. This text is not interpreted or converted to some sort of executable format as typical source code may be. It simply exists to provide information to developers, coders, or users working on or inspecting source code. This kind of text is called a **comment**. We often comment parts of programs to describe what something does, why a choice in construct was chosen, and so on. Typically, a header at the top of our programs is also inserted to provide information such as who authored the program, when it was last updated, and what it does.

In Python, there are two kinds of comments: single- and multi-line comments. Although there are several ways of commenting, we will only discuss the more widely used methods.

Single-line comments span a single line (or a part of a line). A single line comment begins with the *pound* or *hash* (for the Twitter crowd) sign: #. A single-line comments can take up the entire line (i.e., the line begins with #), or it can follow a valid Python statement (i.e., only the latter part of a line is commented). Here are sample single-line comments:

```
# get the user's name
name = input("What is your name? ")
name = "Dr. " + name     # prepend the Dr. title

# say hello!
print "Hello {}!".format(name)
```

In the snippet above, there are three single-line comments. Two each take up an entire line. The third takes up only part of the line. The text that comes before it is valid Python syntax that is interpreted. Note that, once a comment has been started on a line, the rest of the line must be a comment.

Multi-line comments begin and end with three single or double quotes in succession. They are typically used in source code headers, to comment out blocks of code for reasons such as debugging, and so on. Here are sample multi-line comments:

```
"""
Author: Manny McFarlane
Last updated: 2017-11-06
Description: This program is nothing but fluff.
"""

'''
And
here
is
another
multi-line
comment!
'''

""" This is also a valid
      multi-line comment """
```

```
''' And so is this! '''
```

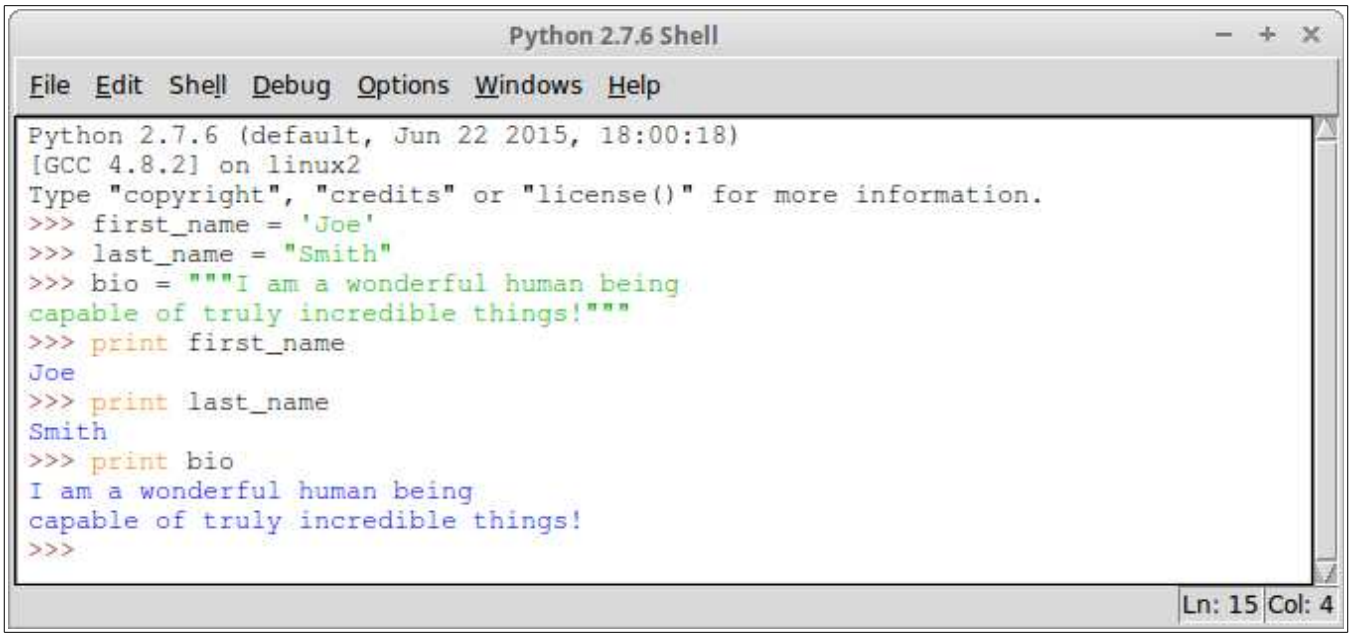
Note that single and double quotes cannot be mixed in multi-line comments. That is, a multi-line comment cannot start with three single quotes and end with three double quotes. Many Python programmers prefer to implement multi-line comments as a sequence of single-line comments; for example:

```
# Author: Manny McFarlane
# Last updated: 2017-11-06
# Description: This program is nothing but fluff.
```

This often stems from the fact that, in Python, strings can be enclosed in single quotes ('), double quotes ("), or three successive single or double quotes. The latter allows strings to span multiple lines. For example:

```
first_name = 'Joe'
last_name = "Smith"
bio = """I am a wonderful human being
capable of truly incredible things!"""
```

Here is the output of this code snippet:



Specifically regarding program headers, many programmers choose to implement them as follows to make them readable and easily identifiable:

```
################################################
# Author: Manny McFarlane
# Last updated: 2017-11-06
# Description: This program is nothing but fluff.
################################################
```

Primary control constructs
What makes computers more than simple calculating devices is their ability to respond in different ways to different situations. In order to create programs capable of solving more complex tasks we need to examine how the basic instructions we have studied can be organized into higher-level constructs. Recall that the vast majority of imperative programming languages support three types of control constructs which are used to group individual statements together and specify the conditions under which they will be executed. Again, these control constructs are: sequence, selection, and repetition.

Recall from a previous RPi activity that **sequence** requires that the individual statements of a program be executed one after another, in the order that they appear in the program. Sequence is defined implicitly by the physical order of the statements. It does not require an explicit program structure. This is related to our previous discussion on **control flow**.

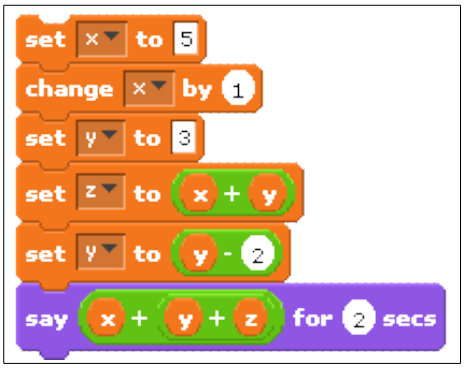
Selection constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed. Basically, selection allows a human programmer to include within a program one or more blocks of *optional* code along with some tests that the program can use to determine which one of the blocks to perform. Selection allows imperative programs to choose which particular set of actions to perform, based on the conditions that exist at the time the construct is encountered during program execution.

Repetition constructs contain exactly one block of statements together with a mechanism for repeating the statements within the block some number of times. There are two major types of repetition: iteration and recursion. **Iteration**, which is usually implemented directly in a programming language as an explicit program structure, often involves repeating a block of statements either (1) while some condition is true or (2) some fixed number of times. **Recursion** involves a subprogram (e.g., a function) that makes reference to itself. As with sequence, recursion does not normally have an explicit program construct associated with it.

Sequence
Sequence is the most basic control construct. It is the *glue* that holds the individual statements of a program together. Yet, when students who are new to programming try to understand how a particular program works, they often just glance over the various statements making up the program to get a *feel* for what it does rather than methodically tracing through the sequence of actions it performs. One reason such an approach is tempting is because students tend to believe they can figure out what a program is *supposed* to do based on contextual clues such as the meaning of variable names and character strings.

While it is often possible to gain a superficial knowledge of a program simply by reading it, this approach will not give you the kind of detailed understanding that is frequently required to accurately predict a program's output. Being able to carefully trace through a program to determine exactly what it does is an important skill. Failure to carefully follow the sequence of instructions often leads to confusion when trying to understand the behavior of a program.

The following Scratch program illustrates the importance of sequence. It contains a little *do nothing* program that displays the value 16. What makes this program interesting is not so much what its output is, as the way in which that output is computed. Without carefully tracing through the program, one statement at a time, it would be difficult to correctly predict the final output generated by the program.



Note that the variables *x*, *y*, and *z* were declared in the variables blocks group. The following illustrates the state of the program's memory after executing each line of code. After performing the first declaration, the program knows only about the variable *x*. After the second declaration, it knows of *x* and *y*, and after the third, it knows of *x*, *y*, and *z*. Since these variables have not yet been assigned values, their values are considered to be undefined at this point.

`declaring x`	*x*					
`declaring y`	*x*		*y*			
`declaring z`	*x*		*y*		*z*	
`set x to 5`	*x*	5	*y*		*z*	
`change x by 1`	*x*	6	*y*		*z*	
`set y to 3`	*x*	6	*y*	3	*z*	
`set z to x + y`	*x*	6	*y*	3	*z*	9
`set y to y - 2`	*x*	6	*y*	1	*z*	9
`say x + y + z for 2 secs`	*x*	6	*y*	1	*z*	9

Program output: `16`

In Python, sequence is implemented in a manner very similar to Scratch: we simply place statements in the order that we wish them to be executed. Here's the program above in Python:

```
x = 5
x += 1
y = 3
z = x + y
y -= 2
print x + y + z
```

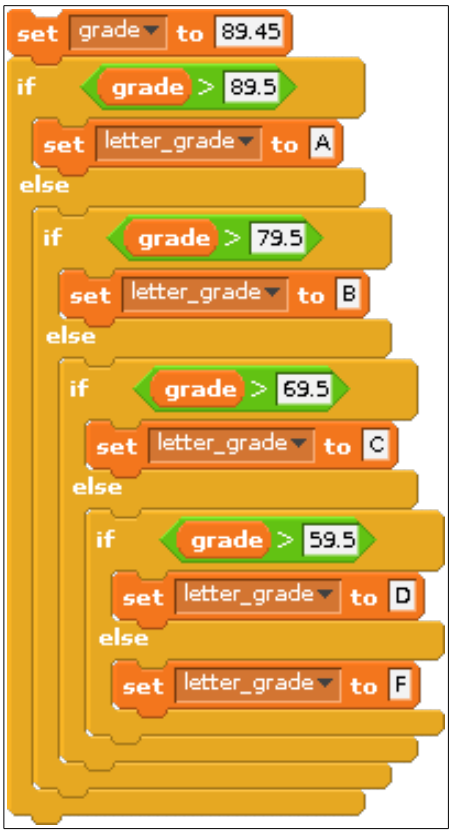
Selection

Selection statements give imperative languages the ability to make choices based on the results of certain condition tests. These condition tests take the form of **Boolean expressions**, which are expressions that evaluate to either *true* or *false*. As discussed earlier, there are various types of Boolean expressions, but most of the time condition tests are based on relational operators. Recall that **relational**

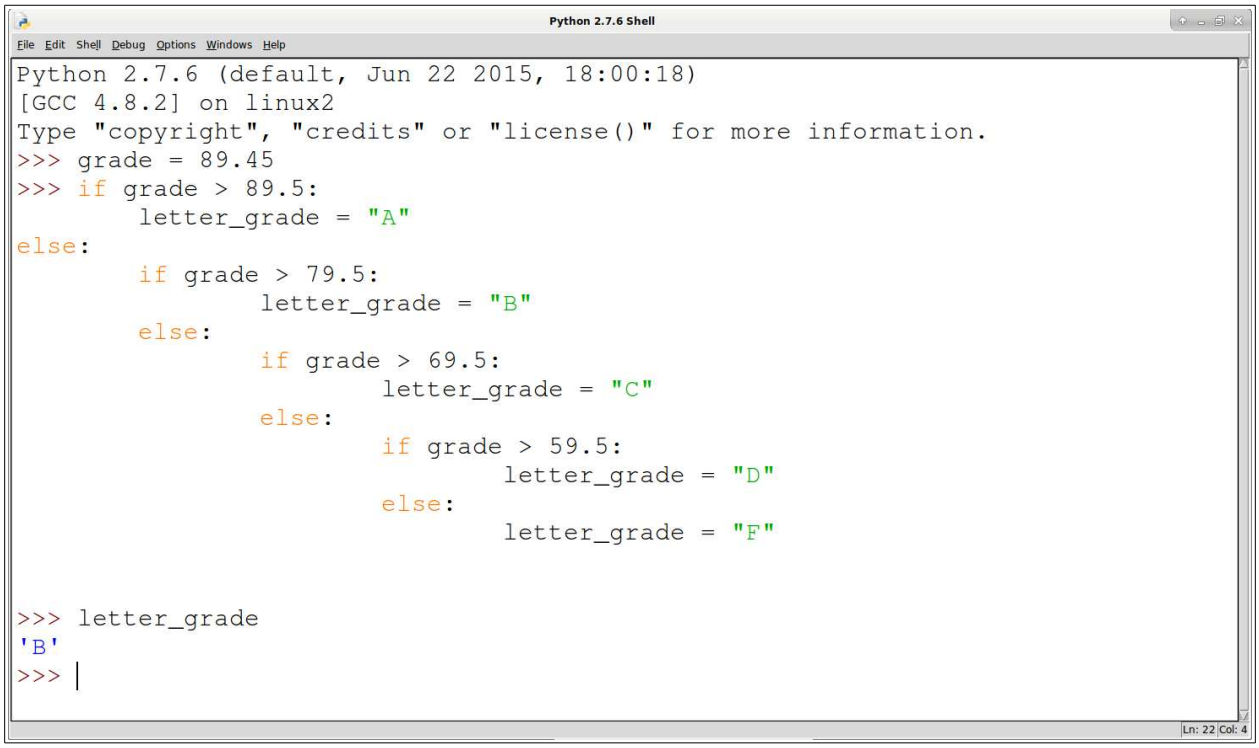
operators compare two expressions of like type to determine whether their values satisfy some criterion. Selection statements use the results of Boolean expressions to choose which sequence of actions to perform next. The general form of all Boolean expressions:

expression relational_operator expression

Both Scratch and Python support two different selection statements: ***if-else*** and ***if***. An *if-else* statement allows a program to make a two-way choice based on the result of a Boolean expression. *If-else* statements specify a Boolean expression and two separate blocks of code: one that is to be executed if the expression is *true*, the other to be executed if the expression is *false*. Recall that, in Scratch, **selection** constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed. Here's an example:



First, convince yourself that the script correctly assigns a letter grade based on a numeric grade. Now here is an equivalent snippet of code in Python:



Note the structure of an *if-else* statement in Python:

```
if condition:
     if_body (true part)
else:
     else_body (false part)
```

Also note the colons after the condition and the **else** reserved word, and the indentation of both the true and false parts. Both are vital in indicating where the *true* and *false* sections of the *if-else* statement begin and end!

Note in the grade/letter_grade example above that there are a few nested *if-else* statements. Python provides a more elegant way to do the same thing using the **elif** clause (which stands for **else if**):

```
if grade > 89.5:
     letter_grade = "A"
elif grade > 79.5:
     letter_grade = "B"
elif grade > 69.5:
     letter_grade = "C"
elif grade > 59.5:
     letter_grade = "D"
else:
     letter_grade = "F"
```

This is indeed a bit more readable.

The *if* statement is similar to the *if-else* statement except that it does not include an *else* block. That is, it only specifies what to do if the Boolean expression is *true*.

The structure of an *if* statement in Python is:

```
if condition:
     if_body (true part)
```

Again, note the colon and indentation. As in the *if-else* statement, both are vital in indicating where the *true* section of the *if* statement begins and ends! *If* statements are generally used by programmers to allow their programs to detect and handle conditions that require *special* or *additional* processing. This is in contrast to *if-else* statements, which can be viewed as selecting between two (or more) equal choices.

Note that the condition of *if* and *if-else* statements can be enclosed in parentheses. This is optional; however, it is recommended as it improves readability:

```
if (condition):
     if_body (true part)
```

For example:

```
if (age > 40):
     print "You are old!"
```

Repetition

Repetition is the name given to the class of control constructs that allow computer programs to repeat a task over and over. This is useful, particularly when considering the idea of solving problems by decomposing them into repeatable steps. There are two primary forms of repetition: *iteration* and *recursion*.

Recall that Scratch supports iteration in two main forms: the *repeat* loop and the *forever* loop. The *repeat* loop has two forms: *repeat-until* and *repeat-n* (where *n* is some fixed or known number of times). The *repeat-until* loop is condition-based; that is, it executes the statements of the loop until a condition becomes true. The *repeat-n* loop is count-based; that is, it executes the statements of the loop *n* times.

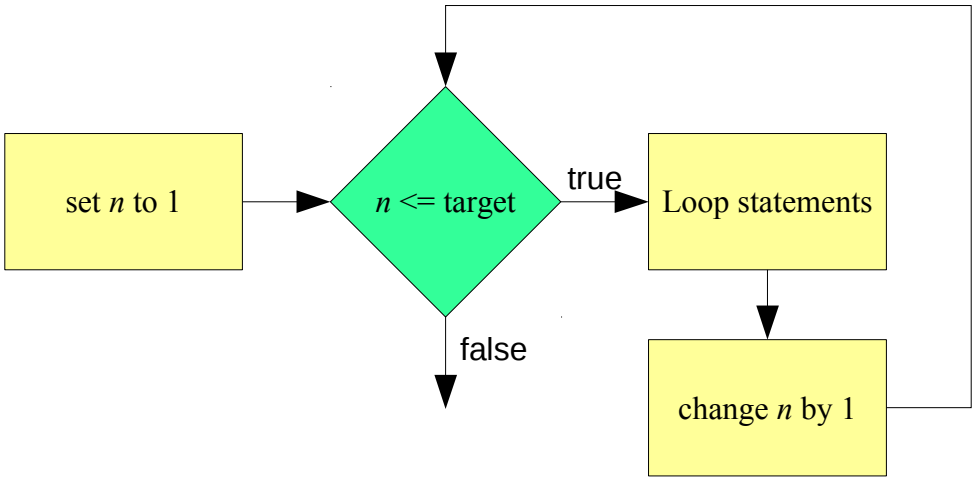
In a *repeat-until* loop, the Boolean expression is first evaluated. If it evaluates to false, the loop statements are executed; otherwise, the loop halts. Here's an example in pseudocode:

```
total ← 0
repeat
     num ← prompt for a positive number (or -1 to quit)
     if num > 0
     then
          total ← total + num
     end
until num < 0
display total
```

This program asks the user to enter a positive number or -1. If a positive number is entered, it is added to a running total. If -1 is entered, the program displays the total and halts. The *repeat-until* loop is used here to repeat the process of asking the user for input until the value entered is less than 0. It is interesting to note that although the program instructs users to enter -1 to quit, the condition that controls

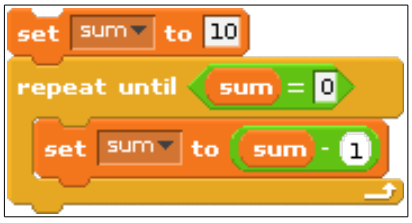
the loop is actually `num < 0` (which will be true for any negative number). Thus, the loop will actually terminate whenever the user enters any number less than zero (e.g., -5).

In Scratch, the *repeat-n* loop executes the loop statements a fixed (or known) number of times. Recall the flowchart for the *repeat-n* loop:



Although the programmer does not have access to a variable that counts the specified number of times (shown as *n* in the figure above), the process works in this manner. A counter is initially set to 1. A Boolean expression is then evaluated that checks to see if that counter is less than or equal to the target value (e.g., 10). If so, the loop statements execute. Once the loop statements have completed, the counter is incremented, and the expression is reevaluated.

Like Scratch, Python provides several constructs for **repetition**. The **while** loop is the most general one, and allows for both event-control (e.g., *repeat-until*) and count-control (e.g., *repeat-n*). Comparing this to Scratch, the while loop is similar to the **repeat-until** and **repeat-n** blocks. Here is a simple example in Scratch:



This simple script initializes a variable, *sum*, to 10. It then repeatedly decrements it by one until it is 0. This can be accomplished in Python using a while loop. The structure of a while loop in Python is:

```
while condition:
    loop_body
```

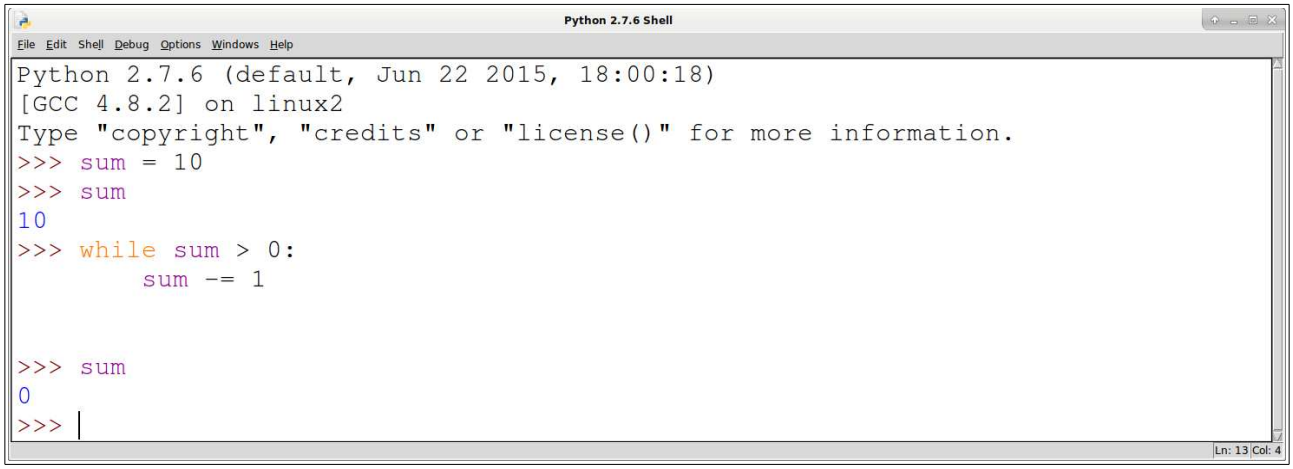
As in if and if-else statements, the condition may be enclosed in parentheses. It is recommended to do so for readability:

```
while (condition):
    loop_body
```

Here is one way to accomplish the same task described in the Scratch script above in Python using a while loop:

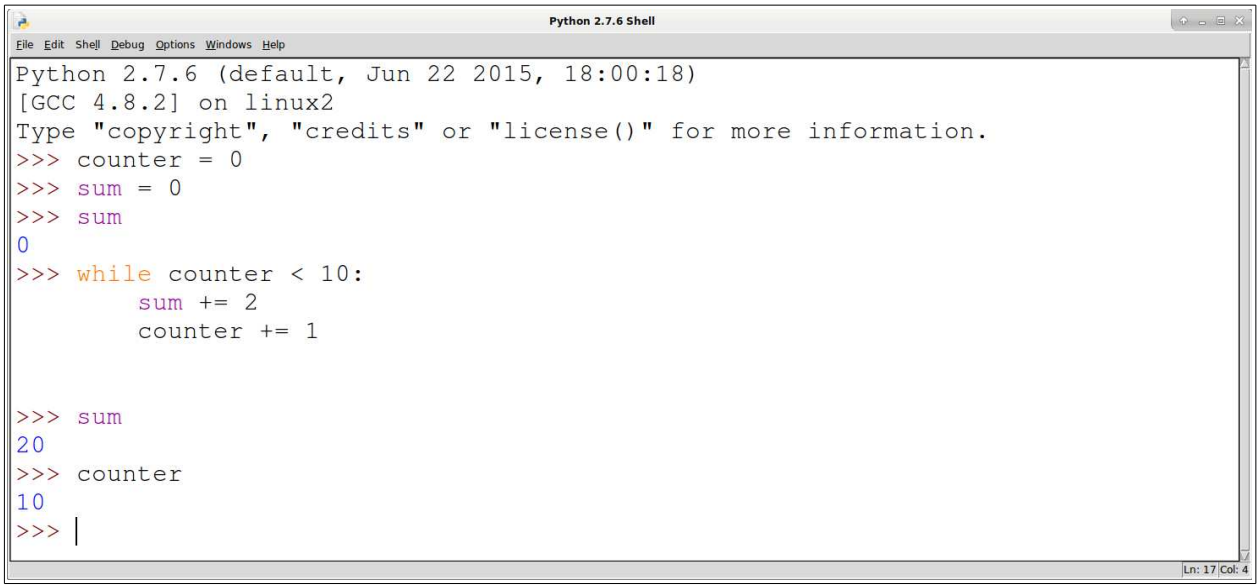
```
sum = 10
while sum > 0:
    sum -= 1
```

Here's this program shown in IDLE:



There is a slight difference between the condition in Scratch's repeat-until loop and the condition in Python's while loop: the condition in the while loop needs to be true to remain in the loop; the loop is terminated whenever the condition evaluates to false. Conversely, the condition in the repeat-until loop should be false to remain in the loop. The repeat-until loop is terminated whenever the condition evaluates to true.

To implement Scratch's **repeat-n** loop in Python with a while loop, we need to create a counter:



Before leaving the topic of iteration, we should say a few words about the idea of *nested* loops. Two loops are nested when one loop appears within the body of another loop. This is quite common, and in fact may be carried out to an arbitrary depth. However, to keep the logic of a program from becoming too hard to follow, programmers try to limit nesting depths to no more than three or four levels deep.

The following Python program displays the multiplication table. This program consists of two nested *repeat-until* loops. The loop variable of the outer loop is `i`, and the loop variable for the inner loop is `j`. Both of these loops count from one to nine:

```
i = 0
while (i < 9):
     i += 1
     j = 0
     while (j < 9):
          j += 1
          print str(i) + " * " + str(j) + " = " + str(i * j)
```

The program's output is of the form `i * j = x`, where `i` and `j` represent the values of the respective variables, and `x` is their product. Notice that `j` runs through its entire range, from 1 to 9, before `i` is incremented by 1. This behavior is easy to understand when you think about the structure of the program.

Let's look at the outer loop. What does it do? Well, first `i` is initialized to 0, it is then compared to 9, and since it is not equal, the first iteration of the loop commences. The first statement of the loop increments `i` and then initializes `j` to 0. The next statement is another *repeat-until* loop. In order for the first iteration of the outer loop to complete, the program must execute this inner loop to completion.

The process repeats until all 81 entries in the multiplication table, from 1×1 to 9×9, are computed and displayed.

We conclude this section with the following Python program:

```
bottles = 99
while (bottles > 0):
     print str(bottles) + " bottles of beer on the wall."
     print str(bottles) + " bottles of beer."
     print "Take one down, pass it around."
     bottles -= 1
     print str(bottles) + " bottles of beer on the wall."
     print
```

This program displays the lyrics to the song, "99 Bottles of Beer on the Wall." As you most likely know, the song begins as follows:

99 bottles of beer on the wall.
99 bottles of beer.
Take one down, pass it around.
98 bottles of beer on the wall.

98 bottles of beer on the wall.
98 bottles of beer.

Take one down, pass it around.
97 bottles of beer on the wall.

It continues in this manner, with one less bottle in each verse, until it finally runs out of beer. An interesting feature of the program is that it decrements *bottles* in the middle of the loop rather than at the end. You should trace through the program with a few bottles to convince yourself that it does work properly. One thing you will probably notice as you do so is that when the program gets down to one beer, it reports that as "1 bottles of beer on the wall." While this lack of grammatical correctness might not seem like such a big deal, especially after 98 beers, we can correct it easily with an *if* statement.

Recursion
Recursion is a type of repetition that is implemented when a subprogram calls itself. When a recursive call takes place, control is passed to what appears to be a brand new copy of the subprogram. This copy of the subprogram may, in turn, call another copy of the subprogram. That copy may call another copy, and so on. Eventually, these recursive calls must terminate and return control to the original calling program.

Take the "99 Bottles of Beer on the Wall" program above. It illustrated an iterative method of singing the song. Here's an example of the same program in Python, implemented using recursion:

```
def consumeBeers(bottles):
     if (bottles > 0):
          print str(bottles) + " bottles of beer on the wall."
          print str(bottles) + " bottles of beer."
          print "Take one down, pass it around."
          print str(bottles - 1) + " bottles of beer on the wall."
          print
          consumeBeers(bottles - 1)

consumeBeers(99)
```

At first glance you might think that this program is nearly identical to the program shown earlier. There are, however, two major differences between the two. First, instead of a *repeat-until* loop, this program has an *if* statement. Also, just before the end of the *if* statement, the same subprogram (*consumeBeers*) is called. This is the recursive call!

Note that the value of the variable *bottles* is decremented by 1 at the recursive call. When control enters the subprogram, the value is checked to see if it is greater than 0. This ensures that, so long as *bottles* is decremented each time the subprogram executes, it will eventually reach 0. When this occurs, it will cause the Boolean expression in the *if* statement to evaluate to false, thereby not executing its block (and calling itself again) and stopping the recursion.

One way to envision recursion is to think of it as a spiral. Each time a subprogram calls itself, we descend down a level of the spiral until we eventually reach the bottom. At that point, execution begins to *unwind* as the subprogram calls complete and we retrace our path back up through the various levels until finally arriving at the *top* level where execution began.

The recursive program above illustrates what is called *tail recursion*, because the recursive call is the last action taken by the subprogram. In tail recursion, there is no work to be done during the *unwinding*

process because it was all done on the way *down* the spiral. In Scratch, this is the only way of implementing recursion. Python permits other forms of recursion, where the recursive call can occur before other statements.

Many students, upon learning how recursion works, worry that programs that employ this form of repetition might be very inefficient in terms of their utilization of machine resources. After all, you have all of those *copies* of the subprogram hanging around. The good news is that recursion is not nearly as expensive as you probably think. For one thing, only one copy of the actual subprogram code is needed. All that is reproduced during each call is the *execution environment*, the variables and whatnot that are used by that *version* of the subprogram. While it is true that recursion generally involves more overhead than iteration, recursive calls are really no more expensive than any other kind of function call. In fact, some optimizing compilers convert tail recursion into iteration so there is often no additional expense in using that form of recursion at all.

Aside from the efficiency issue, you may be wondering why programming languages would support recursion. After all, whenever the need for repetition arises the programmer could always use one of the iteration constructs. The reasons for supporting both recursion and iteration are the same as those, for example, supporting two types of selection statements (*if* and *if-else*): clarity and convenience. Some problems are simply easier to solve using recursion than iteration. For these types of problems, a recursive solution is often more compact and easier to read than an iterative one.

Program flow

It is very important to be able to identify the flow of control in any program, particularly to understand what is going on. In Python, function definitions aren't executed in the order that they are written in the source code. Functions are only executed when they are called. This is perhaps best illustrated with an example:

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

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

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

Each Python statement is numbered for reference. Lines 1 through 5 represent the definition of the function `min`. This function returns the *minimum* of two values provided as parameters. Lines 6 through 10 represent the definition of the function `max`. This function returns the *maximum* of two values provided as parameters. Lines 11 through 14 represent the main part of the program. Although the Python interpreter does *see* lines 1 through 10, those lines are not actually *executed* until the

functions `min` and `max` are actually called. The first line of the program to actually be executed is line 11. In fact, here is the order of the statements executed in this program if *num1* = 34 and *num2* = 55:

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

Let's explain. Line 11 asks the user to provide some value for the first number (which is stored in the variable *num1*). Line 12 asks the user to provide some value for the second number (which is stored in the variable *num2*). Line 13 displays some text; however, part of the text must be obtained by first calling the function `min`. This transfers control to line 1 (where `min` is defined). The two actual parameters, *num1* and *num2*, are then passed in and mapped to the formal parameters defined in `min`, *a* and *b*. Then, line 2 is executed and performs a comparison of the two numbers. Since *a* = 34 and *b* = 55, then the condition in the if-statement is true. Therefore, line 3 is executed before control is transferred back to the main program with the value of the smaller number returned (and then control continues on to line 14). Note that lines 4 and 5 are never executed in this case!

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

What is the order of execution if *num1* = 55 and *num2* = 34? What if *num1* = 100 and *num2* = 100?

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

Computer architecture is a wide branch of computer science that seeks to find answers to questions such as, "What makes up a computer?" and, "How is it that we can use a computer?" The answers to these questions are continuously changing, but we will attempt to give a simple answer in this lesson.

In a previous lesson, we discussed how computer hardware works. Recall that all general-purpose computers, at a minimum, consist of the following hardware components: a central processing unit (CPU), main memory, secondary storage, various input/output (I/O) devices, and a data bus. The **data bus** is like a highway that the other components use to communicate with each other. **Main memory** is used to store data and programs that are currently being used. **I/O devices** allow the outside world to communicate with the computer. The **CPU** is the device that is responsible for actually executing the instructions that make up a program.

Let's further discuss the *brains* of the computer, the CPU. The operation of the CPU is governed by the instruction cycle. The **instruction cycle** is a procedure that consists of three phases: instruction fetch, instruction decode, and instruction execution. The CPU's task is to perform the instruction cycle over and over until explicitly instructed to halt. The **fetch** phase of the instruction cycle consists of retrieving an instruction from memory. The **decode** phase concerns determining what actions the instruction is requesting the CPU to perform. Instruction **execution** involves performing the operation requested by the instruction.

The layers of a computer system

To fully understand computer architecture, it is important to understand the idea of **abstraction** as it is used in the field of computer science. Abstraction is an idea for dealing with complex and interconnected systems whereby a user is only interested in the operations of a certain level of complexity and suppresses more complex details. Abstraction is analogous to looking at Google map of a large country, such as the USA. We can see the individual states, large lakes, surrounding oceans, and neighboring countries. At this level of abstraction, one is unable to see the finer details within a state (such as the names of cities, towns, and major roads). However, zooming in provides an increased level of detail. The entire country is no longer visible; instead, perhaps only a single state (e.g., Louisiana) and its neighbors are visible. At this *zoomed in* level, we can now see some of the cities and major roads. However, we cannot see some of the details of the *zoomed out* level such as the states that are not immediate neighbors or the oceans. If we *zoom in* to an even lower level, we can see street names, major buildings, and so on. Again, we lose some of the details at the higher levels. Dividing a complex system (like a map) into levels that progressively abstract away detail allows users of the system to only deal with information that is relevant at a given time.

A computer is a very complex system consisting of multiple layers (see Figure 1). At the very top is the **user**. Users interact with computers in a variety of ways. That is, they can (and do) interact directly with applications (like a spreadsheet application, a game, or a Web browser). Users can also interact directly with the operating system (e.g., through its GUI or via the console) and with system utilities (think of applications that are provided by the operating system). The **application layer** is the next layer, immediately below the user. It is the layer that a computer user typically interacts with. For example, a user can type and send an email without needing to know how the characters on the screen are made to appear on another computer perhaps one thousand miles away. A user might double-click

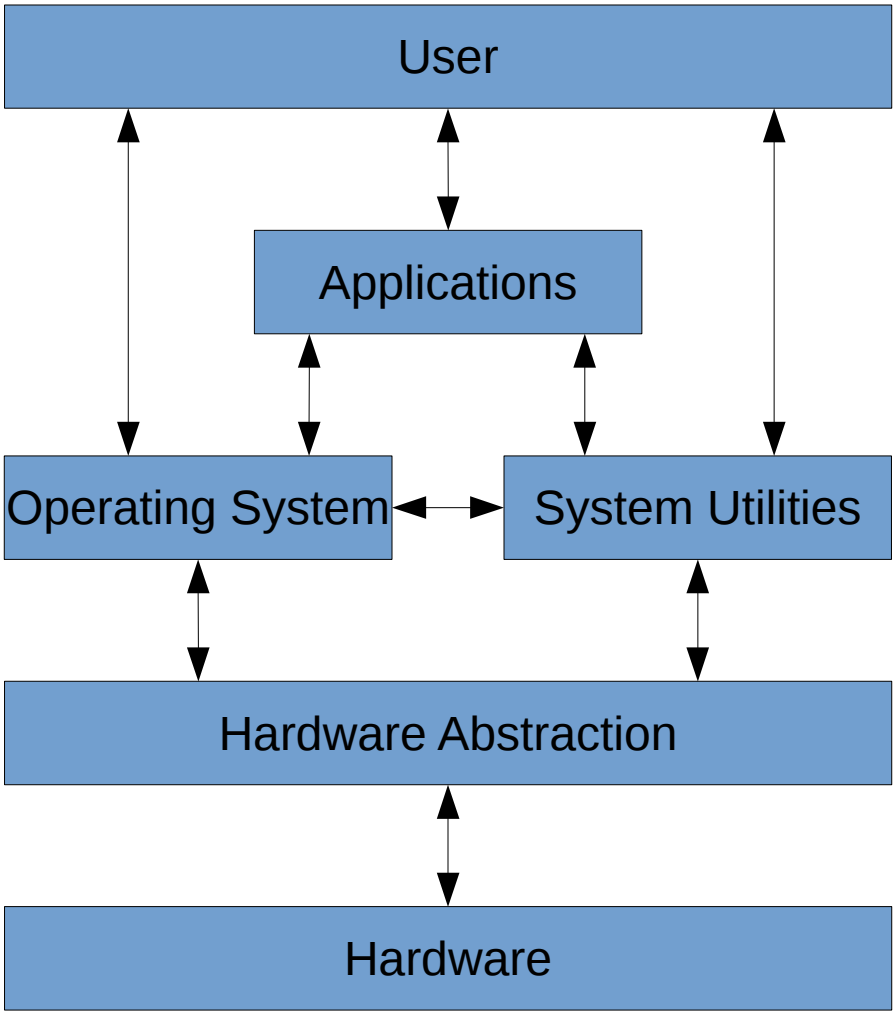


Figure 1: The multiple layers of a computer

an audio file on the desktop without needing to know how the computer understands what a double-click is or how to "play" the audio file.

The next layer is the **operating system** layer. This layer understands user inputs (like typing or double-clicks) and figures out ways of interpreting and executing those inputs. There are many examples of operating systems (e.g., Linux, Windows, MacOS, Unix, Solaris). Of these, Window is still the most common. What is the operating system on your Raspberry Pi? At its core, the operating system is what allows users to interact with the computer and actually make use of it.

System utilities are like applications, but provided directly by the operating system. In one sense, they provide an interface to certain parts of the operating system that allow users to do frequently needed things. For example, the system utility of copying or moving files is often used. Users don't have to install an application that permits copying and moving files around. This is a system utility provided by the operating system. Since system utilities are essentially embedded in the operating system, this layer sits at the same level as the operating system layer.

The layer beneath the operating system layer is the **hardware abstraction** layer (or HAL). Sometimes, this layer is referred to as the device driver layer. There are many different types and designs of computers, and this layer makes sure that the computer hardware acts the same regardless of the

computer's design. For example, it makes sure that the "on" button switches on the computer regardless of where it is located. It makes sure that hitting a specific button opens the CD drive. It provides the operating system with clear instructions on how it can interact with the physical hardware of the computer.

The bottom layer is the **hardware layer**. It represents the physical, tangible stuff that you can see or touch (e.g., keyboard, monitor, mouse, case, power supply, motherboard, etc).

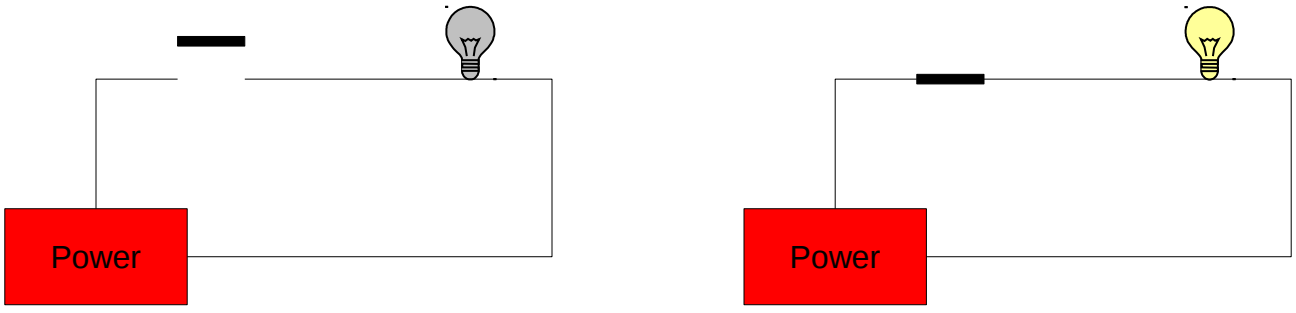
Fundamentals of digital logic

Becoming really good at computer science means having a good understanding of all of the layers, what they do, and how they are used. We will spend most of this lesson dealing with the hardware layer.

A lot of devices have two states: a voltage is high or low, a switch is open or closed, a light is on or off. There are many ways of modeling these two-state systems; some are very concrete and some are more abstract. We'll look at a number of these models, beginning with simple models that are based on mechanical switches and light bulbs.

One of the most basic electrical connection is a light bulb that is either connected to a power source (or not). A slightly more complicated version of this includes a switch that can be either open or closed. These switches are similar to the electrical switches in your home. We will assume that these switches are connected to a source of power that can supply current. The potential of a power source, such as a battery, is called voltage and is measured in units called volts (V). Voltage sources typically have a positive and negative end (called a terminal), and the difference in the potential between both terminals is what we use as the measurement of voltage. Voltage sources can produce either alternating current (AC) or direct current (DC). With DC, one terminal is always positive, and the other is always negative. Examples of DC sources are batteries such as the ones you would put in a small radio, watch, or flashlight. With AC, the two terminals keep on swapping positive and negative roles very quickly (60 times per second!). Examples of AC sources are wall outlets that you would typically find in your home.

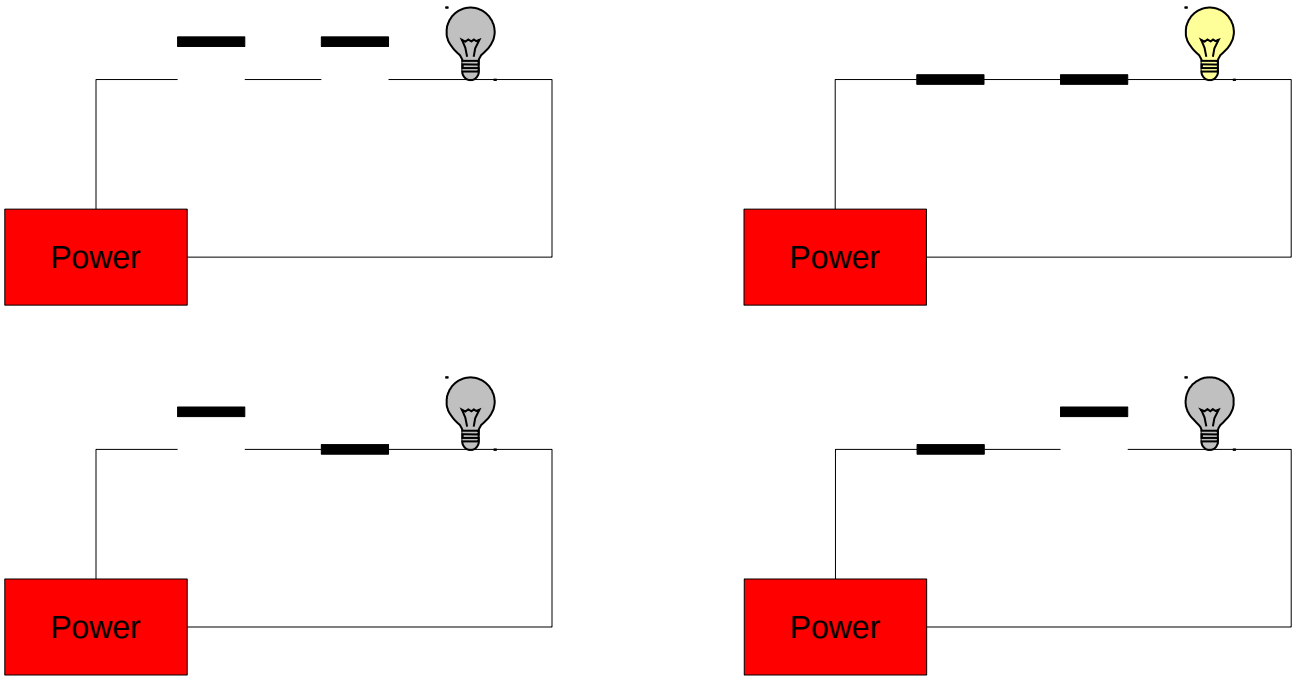
The simplest circuit that can be built contains a power supply, a single switch, and a light bulb. If the switch is open, the light is off; if the switch is closed, the light is on. The following figure illustrates both of these cases:



The state of these two *circuits* can be expressed in table form as follows:

Switch	Light
Open	Off
Closed	**On**

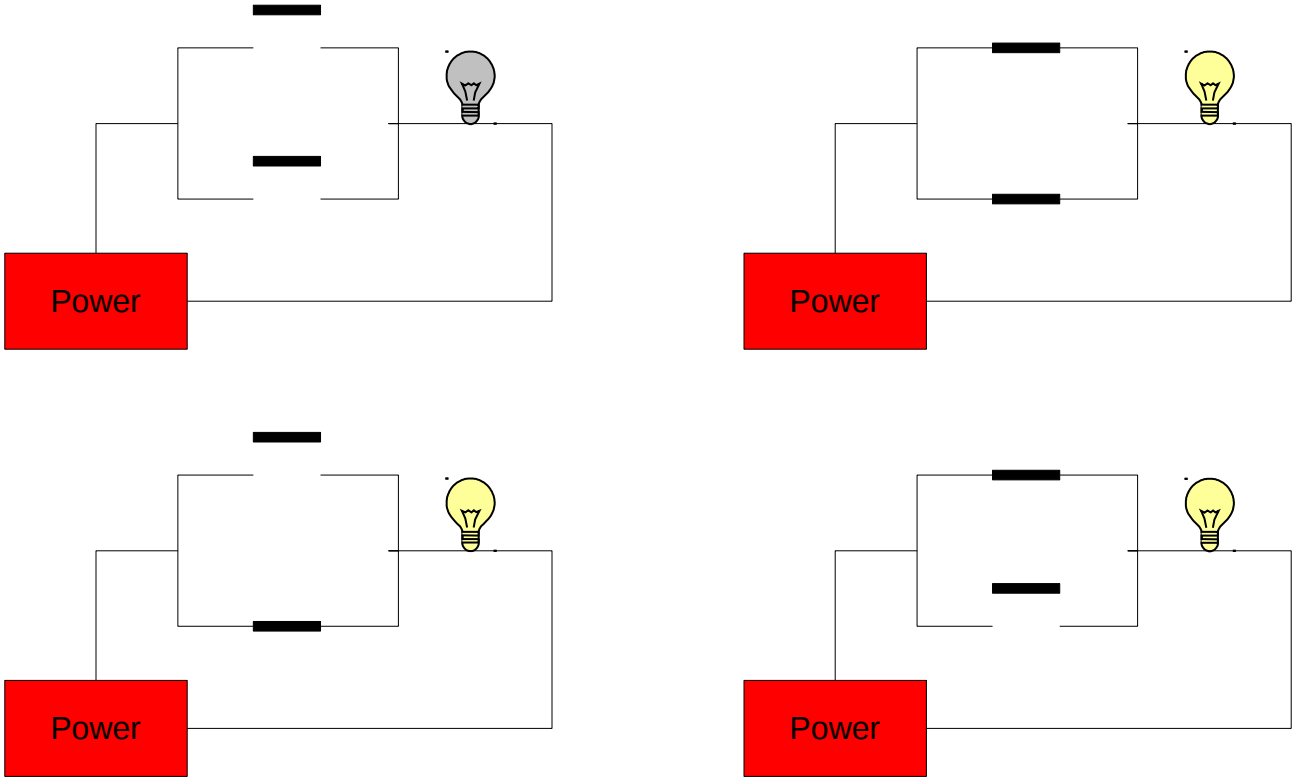
We can increase the complexity of this circuit somewhat by adding a second switch between the first switch and the light bulb. This results in four possible configurations: (1) both switches are open; (2) the first switch is open and the second is closed; (3) the first switch is closed and the second is open; and (4) both switches are closed. This is illustrated in the figure below. These circuits are called **series circuits** since the two switches occur on the same *path* from the power source back to itself. In series circuits, when either or both of the switches are open power will not flow, and the light bulb will be off. Only when both switches are closed does power flow, and the light bulb illuminates. Said another way: if both switch A **and** switch B are closed, then the light will turn on.



The relationship between the states of the two switches (open or closed) and the state of the light bulb (on or off) is summarized in the following table:

Switch A	Switch B	Light
Open	Open	Off
Open	Closed	Off
Closed	Open	Off
Closed	Closed	**On**

Another type of circuit can be designed using two switches. This second type of circuit arranges the switches in parallel rather than in series. In a two-switch **parallel circuit**, each of the switches is placed on a separate path between the power source and the light bulb. The figure below illustrates the four possible configurations of a two-switch parallel circuit. As was the case with the series circuits, there are four possible configurations of the circuit (in fact, they are exactly the same as before). When both switches are open power does not flow and the light bulb is off. However, whenever either or both of the switches are closed, power flows and the light will turn on. Said another way, if switch A **or** switch B is closed, then the light will turn on.



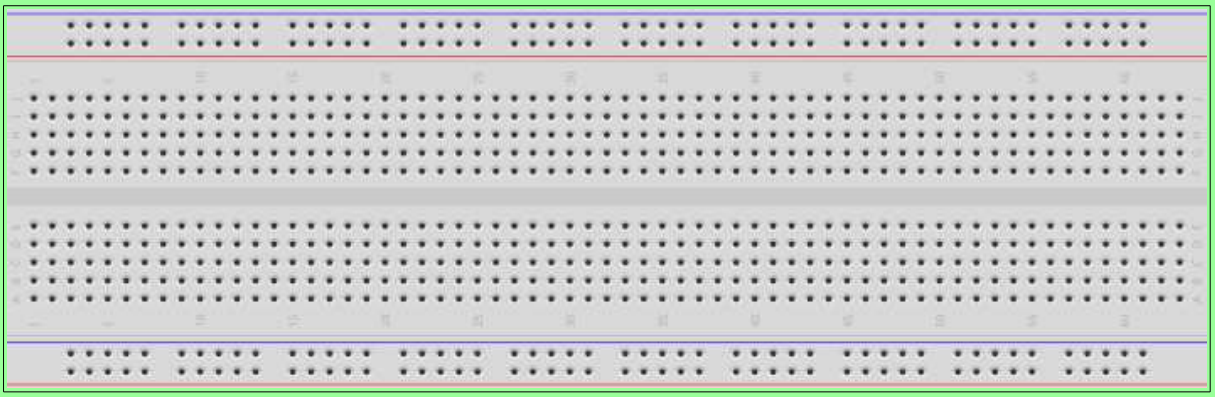
The relationship between the states of the two switches (open or closed) and the state of the light bulb (on or off) is summarized in the following table:

Switch A	Switch B	Light
Open	Open	Off
Open	Closed	**On**
Closed	Open	**On**
Closed	Closed	**On**

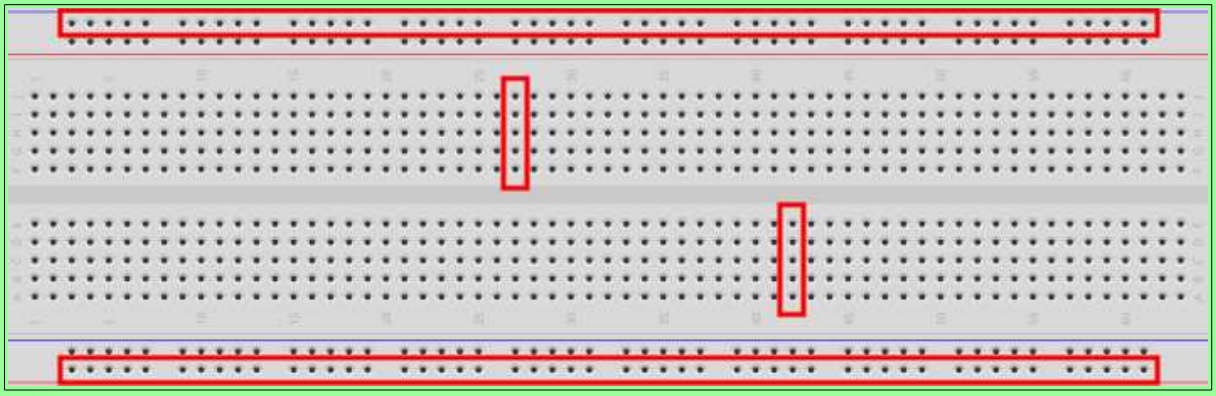
More complex circuits with three or more switches are possible!

Activity 1: LED the Way (preview)

The next Raspberry Pi activity will involve implementing various circuits that illustrate some of the ones covered above. Initially, the Raspberry Pi will only be used as a power source. We will be connecting it to a circuit prototyping board called a **breadboard**, and the Raspberry Pi will provide power to the breadboard. A breadboard is used to simplify the process of prototyping connections between electronic components. It allows the making of secure connections between simple electronic devices by simply plugging them into appropriate rows or columns of the board. Here's an example of a breadboard:



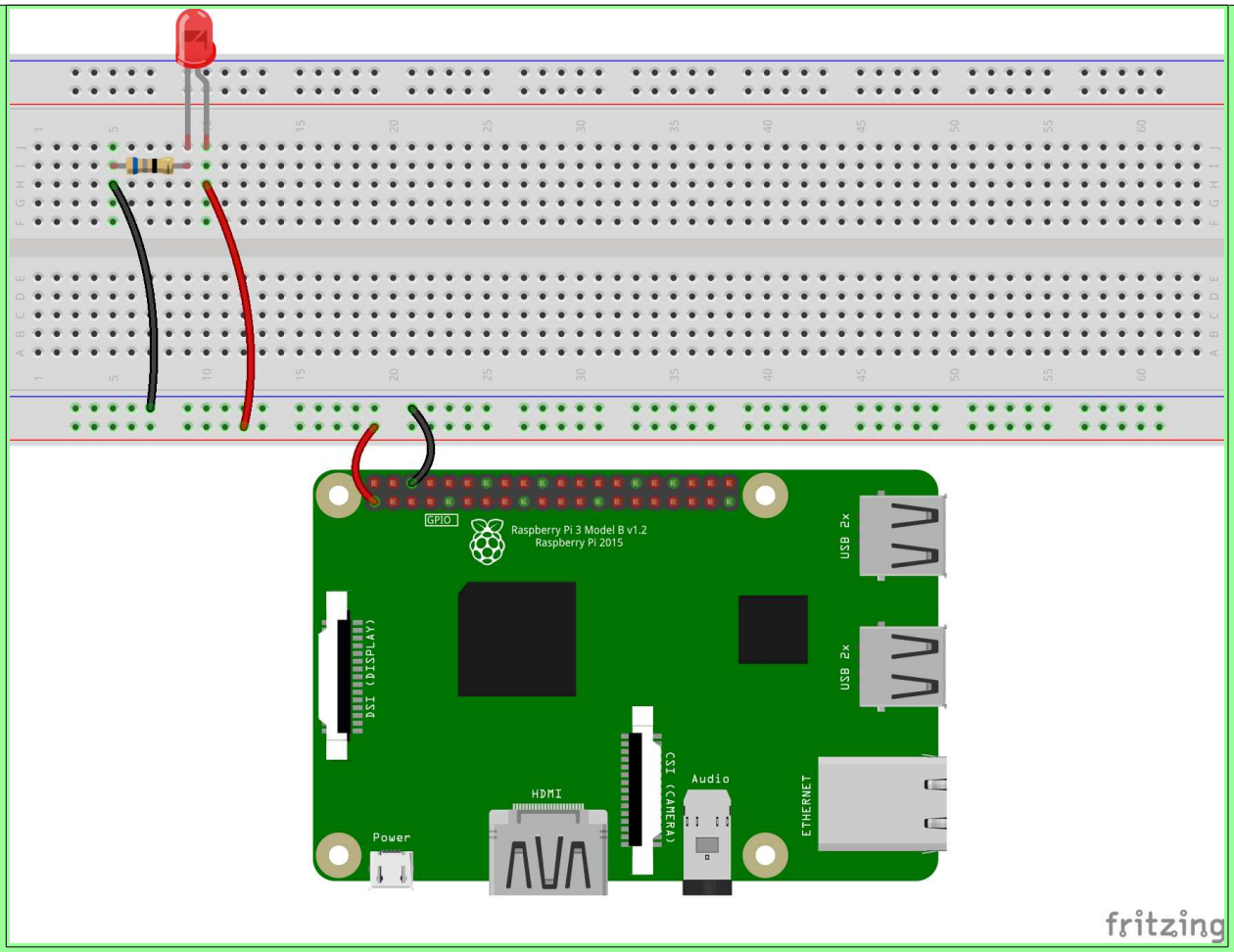
The holes in the breadboard allow electronic components (including wires) to be connected to each other. Note that there are internal connections within the breadboard. Each row along the top and bottom of the breadboard is connected. In addition, each column in the center portion is connected; however, there is a disconnect across the center gap:



The first part of the activity

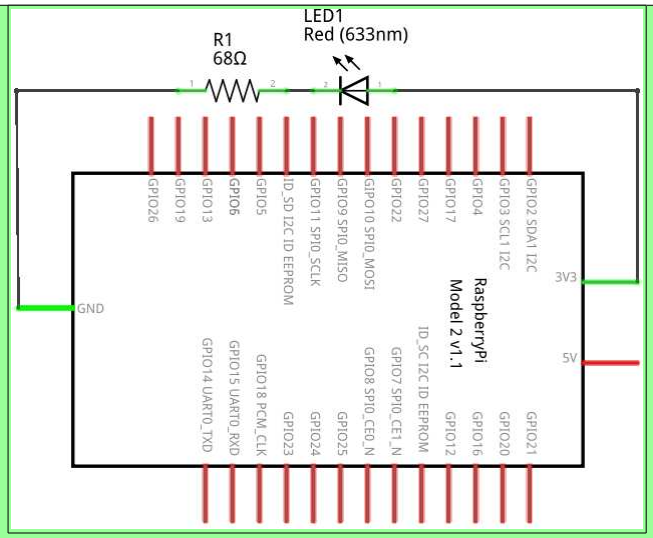
The first part of the Raspberry Pi activity will simply be to connect a power supply to a light. Since the Raspberry Pi provides DC, the light we will use is called an LED. We'll explain this later; but for now, here's an example of the connected electronic components for this part of the activity:

Circuit representation

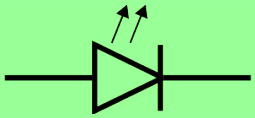


The image above is an example of the topological layout of a circuit. That is, it does a pretty good job of showing how the circuit looks *physically* when connected. Of course, there are many more ways to layout this exact circuit, and this is just one way. This method of diagramming a circuit is called a **layout diagram** because it shows the physical layout of the electronic (and other) components.

A **circuit diagram** (also known as a **schematic**) is another way of representing a circuit that only shows the connections and substitutes actual electronic components with standard symbols. Here's an example of the above circuit as a circuit diagram:



A circuit diagram is a useful way to represent a circuit. Note how it can topologically be laid out in a number of ways. Various electronic components have unique symbols. For example (in the circuit diagram above), the LED has the following symbol:



The resistor has the following symbol:

The large rectangular object with lines coming out of it is the Raspberry Pi. Technically, this represents the GPIO pins on the Raspberry Pi. We'll discuss this more later. We will also show more electronic components and their symbols later.

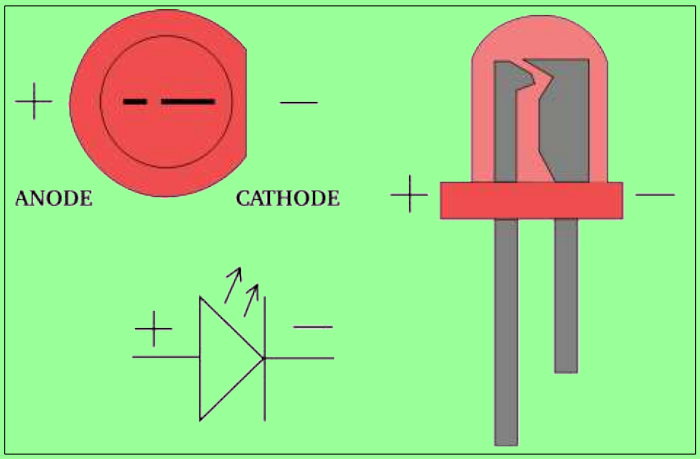
The components

Let's go through the components, one-by-one. At the bottom is the Raspberry Pi. You will notice that there are two wires connecting some pins on the RPi to the breadboard. We typically use red wires to signify positive voltage and black wires to signify negative voltage. In DC, the negative side is called ground. So red wires connect positive power to something, and black wires connect something to ground.

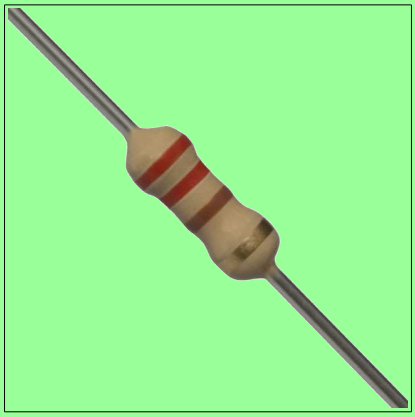
The red "light" in the circuit is called an **LED** (Light Emitting Diode). An LED is more convenient than a traditional light bulb, because it does not require high voltage in order to turn it on. In fact, it consumes such a small voltage that typical higher voltage levels would render the LED unusable. Be careful when using LEDs, and never connect them directly to a voltage source.

An LED allows current to flow through it in only one direction (from positive to negative). LEDs have a short leg and a long leg. The short leg is called the **cathode** and is the *negative* side. The long leg is

called the **anode** and is the *positive* side. The head of an LED is also flat on one side: the negative (or cathode) side. LEDs come in various colors (the one in the circuit above is red, for example). The longer leg of an LED should **always** be connected to the positive side of your voltage source. If it is connected backwards (i.e., with the shorter leg connected to the positive side), the LED will not light and may even burn out. For this reason, an LED should always be connected to a DC voltage source.



Since most power sources are too strong for typical LEDs, we must reduce the current somewhat so that the LED does not become damaged. **Resistors** are typically used to resist the flow of electricity. When using them with LEDs, we typically connect a resistor in *series* with the LED. It doesn't matter if the resistor is on the positive or negative side of the LED. It works the same in either case. Resistors come in various resistances. Resistance is measured in a unit called the ohm (Ω). Here is an example of a 220Ω resistor:



We can calculate the resistance required to resist the flow of electricity through the LED using Ohm's Law. Ohm's Law establishes a relationship between voltage, current, and resistance. Let's first fully define each of these:

> **Voltage** is the difference in electric potential energy between two points. It can be considered as electric *pressure* and/or the work required to move electric charge between two points. The unit used to represent voltage is the **volt** (V).
>
> **Current** is the flow of electric charge (or electrons moving through a wire). The unit used to

represent current is the **ampere** (A), or **amps**. We typically used the symbol I to represent current in a mathematical formula (such as Ohm's Law).

Resistance is the measure of difficulty to pass an electric current through a conductor. A conductor is some material that allows the flow of electric current. The unit used to represent resistance is the ohm (Ω). We typically used the symbol R to represent resistance in a mathematical formula (such as Ohm's Law).

Ohm's Law is defined as the following:

$$V = IR$$

Stated formally, the voltage (electric potential difference) across two points on a circuit is equivalent to the product of the current between those two points and the total resistance of all electrical devices present between those two points.

Consider the LED circuit above, where the red LED requires a forward voltage of 2V (i.e., the amount of voltage required across the LED to light it) and has a forward current of 20mA (i.e., the amount of current flow required through the LED to sufficiently power it on). These values are provided in the data sheet of the LED. A **data sheet** is a document that provides technical information about an electrical component.

We can calculate the resistance required in the circuit to ensure that the LED lights up properly and is not possibly damaged by having too much current move through it or too much voltage across it. Suppose that our power source (the Raspberry Pi) provides 3.3V. The voltage difference across the source voltage and ground is 3.3V (since ground is at 0V). According to the data sheet, the LED requires 2V across its *legs* and requires 20mA of current through it. Using Ohm's Law we can solve for R. The value for V is 1.3V (3.3V at the source – 2V through the LED), and the value for I is 0.02A (20mA required through the LED). And now we solve:

$$\begin{aligned} V &= I * R \\ (3.3V - 2V) &= 0.02A * R \\ 1.3V &= 0.02A * R \\ 65 &= R \end{aligned}$$

So the resistance should be 65Ω. The closest valued resistor available is 68Ω. We can therefore use a 68Ω resistor in series with the LED. This should be sufficient to turn it on brightly without damaging it.

You may have noticed that resistors also have a *wattage* rating. To explain this, we must first discuss electric power. Electric power is the rate at which electric energy is transferred by a circuit. The unit used to represent power is the **watt** (W). Each component in a circuit dissipates power (as heat – usually through friction – as electrons move through the component). Therefore, each component has a power rating that provides a measure of how much power it can dissipate without breaking down. We can calculate the power dissipated in a circuit using a variant of Ohm's Law:

$$P = VI$$

The power in a circuit is defined as the product of the voltage across two points on a circuit and the current between those two points. In the LED example above, the total power dissipated in the circuit is calculated as follows:

$$\begin{aligned} P &= V & * I \\ P &= 3.3V & * 0.02A \\ P &= 0.066W \end{aligned}$$

To calculate the power dissipated by each component, we simply need to isolate the voltage drop across each. The current is constant in the entire circuit. So for the LED, we can calculate the power dissipated as follows:

$$\begin{aligned} P &= V & * I \\ P &= 2V & * 0.02A \\ P &= 0.04W \end{aligned}$$

So we would need an LED rated at 0.04W. And for the resistor:

$$\begin{aligned} P &= V & * I \\ P &= (3.3V - 2V) & * 0.02A \\ P &= 1.3V & * 0.02A \\ P &= 0.026W \end{aligned}$$

So we would need a resistor rated at 0.026W.

In the end, we usually opt for a power rating that is greater than the actual power dissipated by the component (so that it can last a long time). A good target is not to exceed 60% of the wattage rating of the component. For the resistor, this means a power rating of 0.043W (0.026W / 0.6). Most typical resistors are rated at 0.25W (some are 0.125W and others are much higher). For the LED, this means a power rating of 0.067W (0.04W / 0.6), or 67mW. Most typical LEDs are rated at approximately 120mW. For this circuit, a typical LED rated at 120mW and a resistor rated at 1/8W would work just fine.

Did you know?

Resistors have different values, and the value of a resistor can be determined by looking at the colored *bands* that surround its body. Because resistors are typically small in size (any letters written on one would be too small to be easily read), engineers invented a color code that can be used to calculate the resistance of a resistor. There are multiple online resources that can teach you how to read the value of a resistor from its colors.

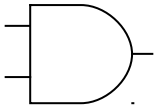
Did you know?

Breadboards actually derive their name from a breadboard (i.e., a wooden board on which bread is often cut). This is because early versions of breadboards were made from the wooden bread cutting workstations.

Logic gates

Gates are electronic versions of the mechanical switches introduced earlier. Some gates have multiple inputs, but all gates have a single output. Just as the switches and light bulbs of the previous examples were always in either of two states, the inputs and outputs of gates are confined to two voltage states. The voltage of every input to the gate, as well as the output from the gate, must be either high (positive voltage) or low (0V, or ground). We use the symbol "1" to represent the high voltage state and "0" to represent the low voltage state.

There are three basic kinds of logic gates: *and* gates, *or* gates, and *not* gates. An ***and*** gate has two inputs and one output. The output is "1" (high) only when both inputs are "1" (high). In all other cases the output of *and* is "0" (low). Here is the symbol for an *and* gate (where the two inputs are on the left, and the output is on the right):

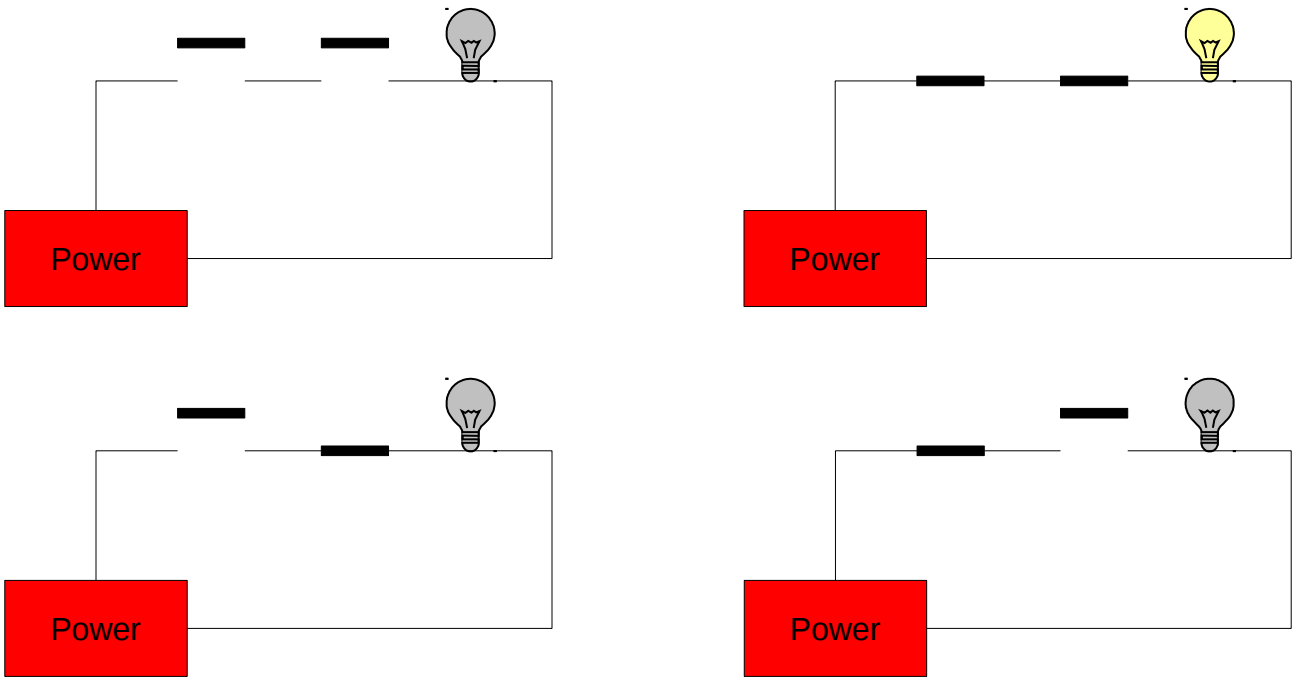


We can represent the possible states of a gate in a truth table. A **truth table** defines the meaning of a gate, or circuit, by listing every possible configuration of inputs along with the corresponding output. Traditionally, inputs are listed on the left side of the table with the output on the right. Each row of the truth table represents one configuration that the circuit can be in. Truth tables for circuits with n inputs will always have exactly 2^n rows, one for each possible configuration of the inputs. The following is the truth table for the *and* gate, where the inputs have been labeled A and B, and the output has been labeled Z:

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

Since the *and* gate has two inputs, its truth table will contain $2^2 = 4$ rows. The first row of the truth table represents the situation in which both inputs to the *and* gate are low. In this case the output will be low as well. The second and third rows cover the cases in which one of the inputs is high and the other is low. In line two, the first input is low and the second is high; whereas in line three, the first input is high and the second is low. In either case, the output is low. The final row of the table represents the situation in which both inputs are high. In this case, the output will be high as well.

The functionality of the *and* gate can be implemented by the series circuit introduced earlier:



If the switches represent the inputs, *A* and *B*, then this circuit correctly produces the output, Z, of an *and* gate (which is the light bulb in the circuit). In fact, compare the truth table for the *and* gate above with the truth table for the circuit:

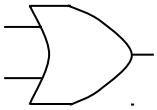
Two Switches in Series Circuit		
Switch A	**Switch B**	**Light**
Open	Open	Off
Open	Closed	Off
Closed	Open	Off
Closed	Closed	**On**

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

If *Open* is replaced with 0 and *Closed* with 1, the tables are the same. The reason that truth tables are called as such is that if 1 is taken to mean *true* and 0 is taken to mean *false*, then the output of the table defines the circumstances under which the specified logical operation is true. For example, in common English usage, *A and B* will be true only when both *A* and *B* are true. The statement: "My cat is old and fat" is only true when the cat in question is both "old" and "fat." If my pet cat were either young, or skinny, or both, then the statement would be false.

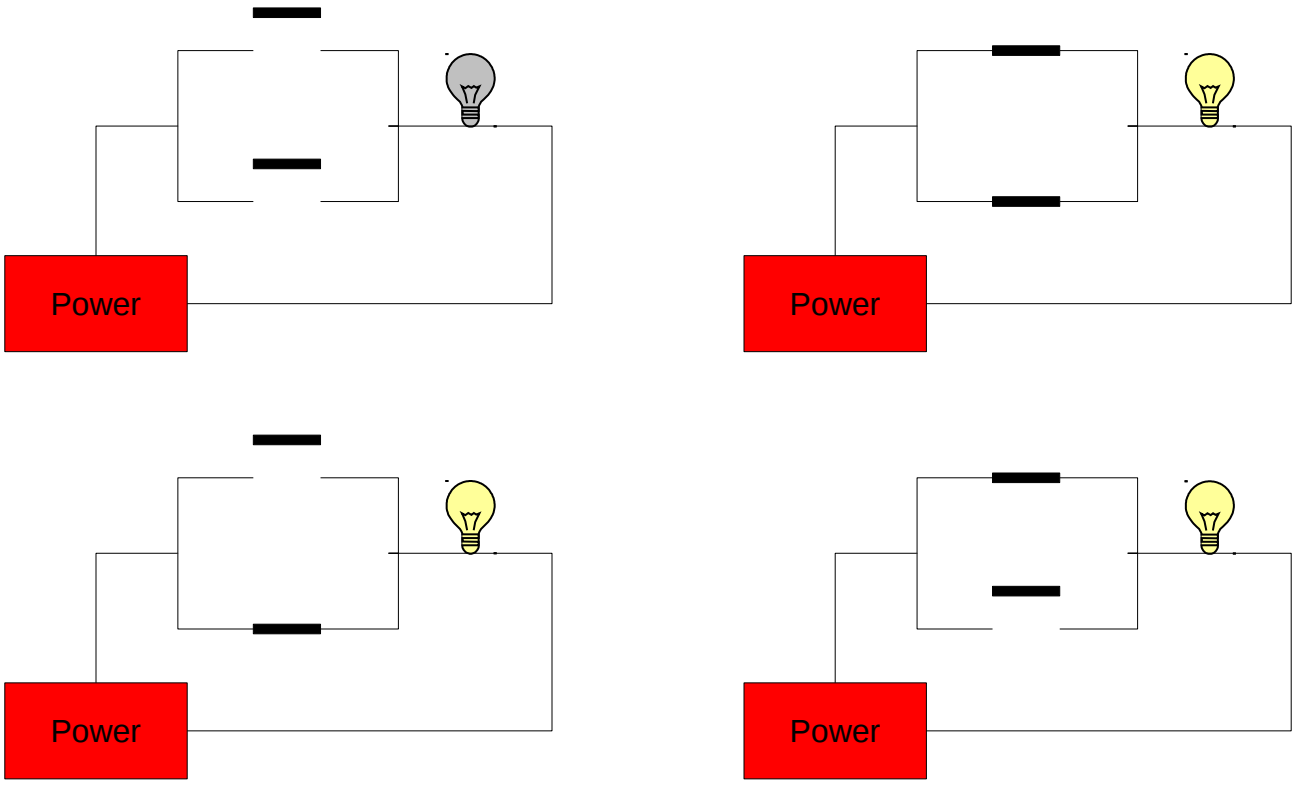
The thing that is so exceedingly cool about logic gates, and the circuits that implement them, is that very simple devices can capture small parts of what humans consider logical reasoning. As you can well imagine, this idea caused great excitement when first discovered.

The ***or*** gate is similar to an *and* gate, in that it has two inputs and one output. The output of the *or* gate is 1 whenever either (or both) of the inputs are 1. The only case in which the output is 0 is when both of the inputs are 0. Here is the symbol and truth table for the *or* gate:



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

Again, the two inputs of the *or* gate are labeled *A* and *B*, and its output is labeled *Z*. Notice that the *or* gate can be implemented by the parallel circuit introduced earlier:

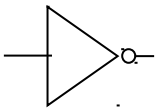


Two Switches in Parallel Circuit		
Switch A	**Switch B**	**Light**
Open	Open	Off
Open	Closed	**On**
Closed	Open	**On**
Closed	Closed	**On**

OR Gate		
A	**B**	**Z**
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

You should convince yourself that the behavior of the *or* gate captures the semantics of the word "or" as it is commonly used. The statement: "My cat is either on the couch or under the bed" is true if either the phrase "My cat is on the couch" is true or the phrase "my cat is under the bed" is true. The original statement is false only when neither of these phrases is true.

The third basic logic gate is the *not* gate. The ***not*** gate has a single input and a single output. The output is the inverse of the input. Here is the symbol and truth table for the *not* gate:



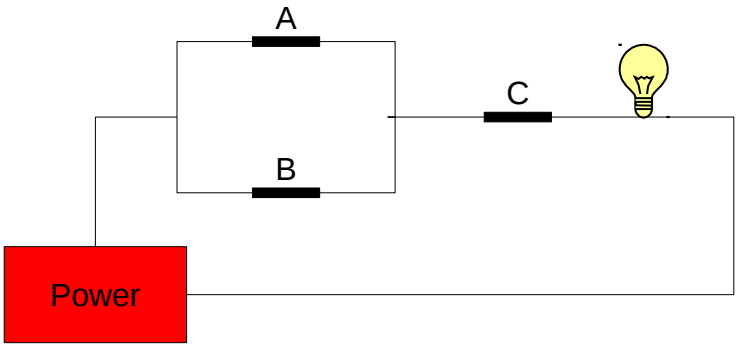
A	Z
0	**1**
1	0

Note that this truth table consists of only two rows rather than four (as was the case with the *and* and *or* gates). This is consistent with the claim that truth tables contain exactly 2^n rows for an n input circuit. Since the *not* gate takes in only a single input, there are only two possible configurations that the gate can be in.

As with the *and* and *or* gates, the behavior of the *not* gate captures the semantics of the word. If the sentence: "My cat is black" is true, then the sentence "My cat is *not* black" would be false (and vice versa).

Combining gates

Consider the following circuit:

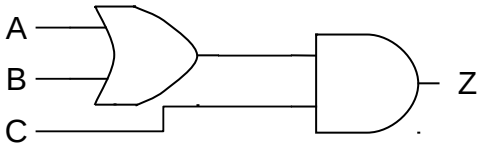


Here's its truth table:

Switch A	Switch B	Switch C	Light
Open	Open	Open	Off
Open	Open	Closed	Off
Open	Closed	Open	Off
Open	Closed	Closed	**On**
Closed	Open	Open	Off
Closed	Open	Closed	**On**
Closed	Closed	Open	Off
Closed	Closed	Closed	**On**

So long as either *A* ***or*** *B* is closed ***and*** *C* is closed, then the light bulb is lit. *C* must be closed in order for the bulb to be lit.

It is natural to ask at this point what an equivalent circuit consisting of logic gates would look like. Since switches *A* and *B* are in parallel, this portion of the circuit can be represented using an *or* gate. The output of that part of the circuit is in series with *C*, so it can be modeled with an *and* gate. The logic gate circuit shown below is thus equivalent to the switch circuit given above.



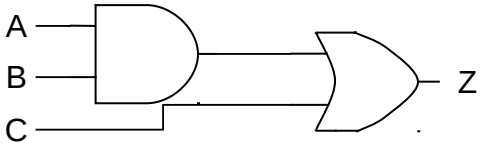
In fact, here is the truth table for this circuit:

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	**1**
1	0	0	0
1	0	1	**1**
1	1	0	0
1	1	1	**1**

For readability and to make it a bit easier to derive, we can expand the truth table to provide intermediate gate outputs as follows (where *Z* is the output of *A or B*, and *Z'* is the output of *Z and C*):

A	B	Z	C	Z'
0	0	0	0	0
0	0	0	1	0
0	1	1	0	0
0	1	1	1	**1**
1	0	1	0	0
1	0	1	1	**1**
1	1	1	0	0
1	1	1	1	**1**

Can you fill in the truth table for the circuit below? Let *Z* represent the output of *A and B*, and *Z'* represent the output of *Z or C*.



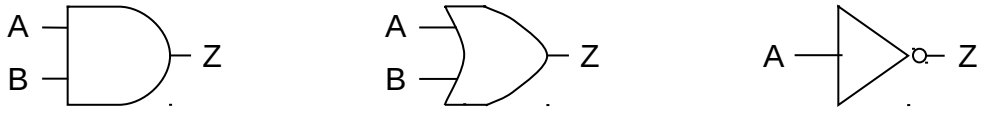
A	B	Z	C	Z'

Boolean algebra

The arithmetic that is used to reason about two-state systems was first developed by George Boole in 1854. **Boolean algebra** is a mathematics based on three fundamental operators: *and*, *or*, and *not*; and the variables on which they operate. Boolean variables are binary, having only two valid states: 1 (representing *true*) and 0 (representing *false*).

The operator *and* is written as a dot " $\cdot$ ", *or* is written as a plus "+", and *not* is written as a horizontal bar drawn over the expression being negated. The behavior of these three Boolean operators is identical to the behavior of the corresponding logic gates. Thus, the expression $A \cdot B$, meaning *A and B*, will be 1 (true) when the variables *A* and *B* are both 1 (true). The expression $A + B$, meaning *A or B*, will be 1

when either or both variables are 1. The expression *not A* (written $\overline{A}$), will be 0 when *A* is 1 and 1 when *A* is 0. The relationship between the Boolean operators and the fundamental logic gates is illustrated below. In the illustration, the Boolean variables *A* and *B* correspond to the inputs to the circuit, and the variable *Z* corresponds to the output.

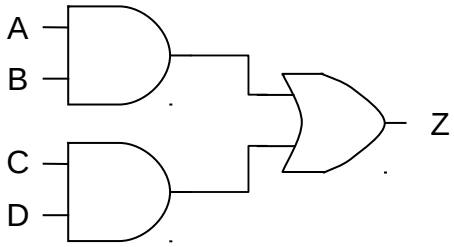


$Z = A \cdot B$ $Z = A + B$ $Z = \overline{A}$

As in ordinary algebra, Boolean algebra uses parentheses to indicate which operands go with which operators. The Boolean expression $A + (B \cdot C)$ represents a completely different circuit from $(A + B) \cdot C$. In the first, *B* and *C* are fed into an *and* gate, with the result being sent (along with *A*) into an *or* gate. In the second, *A* and *B* are fed into an *or* gate, with the result being combined with *C* via an *and* gate.

As you may be beginning to suspect, there is a direct correspondence between Boolean expressions and logic circuits. Every logic circuit that can ever be constructed will have a corresponding Boolean expression, and every valid Boolean expression that can ever be written maps to an equivalent logic circuit. The process of converting between the two representations is quite mechanical: simply use the substitutions above, being sure to parenthesize Boolean expressions in a manner that preserves which operators go with which operands.

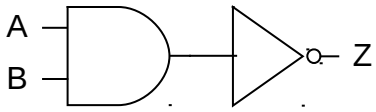
Try to write the Boolean expression corresponding to the following circuit in the space below:



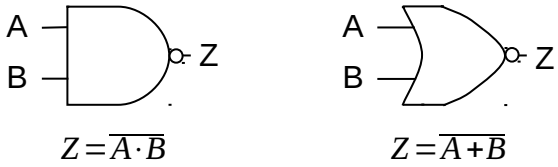
Boolean algebra provides computer scientists and engineers a powerful tool for concisely representing circuits and reasoning about their behavior. While the details are beyond the scope of this lesson, Boolean algebra allows us to do things like prove that two different circuits compute the same function; or find simpler (and thus less expensive) ways of implementing the functionality of a circuit.

Other gates

Any device, whose operation can be defined in terms of a truth table or Boolean expression, can be implemented using only the fundamental logic gates: *and*, *or*, and *not*. However, a number of additional gates are usually defined, as they prove useful for practical purposes. For example, it is frequently the case that a *not* will immediately follow an *and* gate, like so:

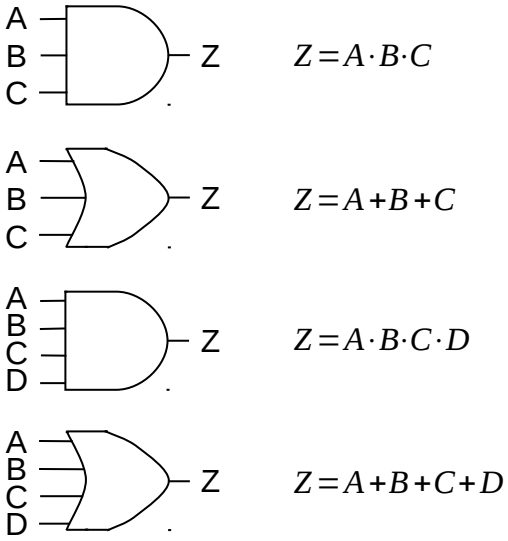


Since this is such a common occurrence, the circuit has been given a name (*nand*) and a gate symbol (the *and* symbol combined with the bubble from the *not* symbol). Similarly, *not* often follows *or*, so there is a *nor* gate whose symbol is the bubble from the *not* attached to the *or* symbol. The following figure illustrates both the *nand* and *nor* gates. Their behavior, in terms of Boolean expressions, is provided as well. It is important to remember that these gates are simply a convenience (a kind of *shorthand*), in that they allow a circuit to be constructed from fewer underlying components.

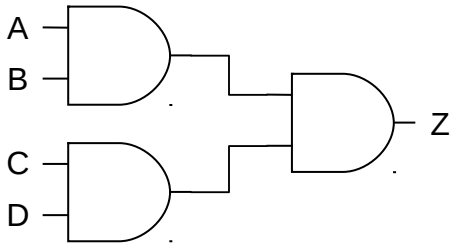


$Z = \overline{A \cdot B}$ $Z = \overline{A + B}$

As another example, the basic *and* and *or* gates support only two inputs; however, a circuit designer will frequently need to *and* or *or* more than two inputs. For this reason multi-input *and* and *or* gates exist. The following figure presents the three and four input *and* and *or* gates along with their Boolean expressions:



While these gates are often quite convenient, remember that it is always possible to construct equivalent circuits from the underlying two-input gates. For example, the following circuit represents one possible implementation of a four-input *and* gate:



Its Boolean expression is $Z = (A \cdot B) \cdot (C \cdot D)$. Note, however, that it could be designed differently (with a different Boolean expression), yet still represent a four-input *and* gate. For example, $Z = ((A \cdot B) \cdot C) \cdot D$ would also work. The other multi-input gates can be constructed in a similar manner.

In addition to multi-input *and* and *or* gates, multi-input *nand* and *nor* gates can be constructed. The symbols for these gates are identical to the symbols for the multi-input *and* and *or* gates, with the exception of a *not* bubble attached to the output of each gate symbol. Their Boolean expressions are also identical as well, except that a *not* bar appears above the right-hand side of the expression.

Combinational circuits

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. In this lesson, we will focus on the simplest combinational circuits. Let's start with a relatively simple circuit, the *exclusive or*.

An exclusive or, or *xor*, has two inputs and a single output. Its behavior is defined by the following truth table, where the inputs are labeled *A* and *B* and the output is labeled *Z*:

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

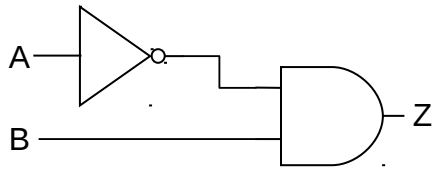
Like the standard two-input *or*, the *xor* produces a 1 (true) when either of its inputs are 1, and a 0 (false) when both of its inputs are 0. The difference between *or* and *xor* appears in the case when both inputs are 1. The standard *or* produces a 1 in this case. The *xor* generates a 0. In other words, the "exclusive or" outputs a 1 when either, *but not both*, of its inputs are 1.

English does not contain a unique word for expressing the idea of *xor* – the word "or" does double duty for both its "inclusive" and "exclusive" forms. However, one can usually tell from the context of a sentence which form is intended. For example, if you tell a child "you can have candy or popcorn," the intended meaning is exclusive or – either candy or popcorn, but not both. On the other hand, if a friend says "I'd be happy winning either the Porsche or the Mercedes," the intended meaning is inclusive or – you would certainly not expect your friend to become unhappy if he won both cars.

Now that we understand the behavior of *xor* in terms of its inputs and outputs, we can turn our attention to the problem of designing a circuit with its behavior. But how are we to begin?

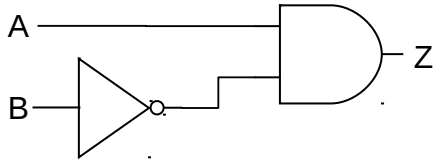
One approach that often gets you moving in the right direction is to examine the truth table to determine the various circumstances under which the circuit must produce a 1. In the case of *xor*, there are two such cases: one in which input *A* is 0 and input *B* is 1, and another in which input *A* is 1 and input *B* is 0. Once these cases have been identified, we proceed by designing *sub-circuits* that will produce 1 in each of the required cases. The final step is to combine the sub-circuits together using an *or* gate. This is necessary because the main circuit would be true under any of the cases in which the sub-circuits generate a 1.

The following sub-circuit will generate a 1 when input *A* is 0 and input *B* is 1. Its Boolean expression is $Z = \overline{A} \cdot B$:

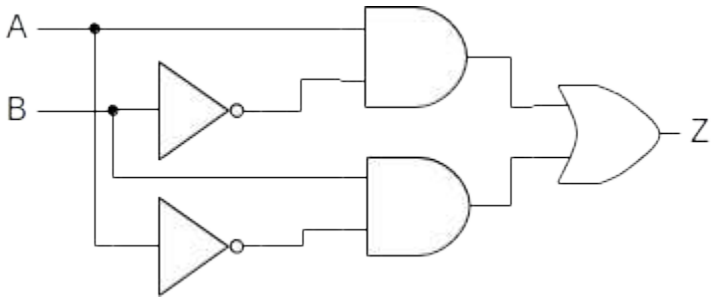


It works by negating *A* and feeding that result (together with *B*) into an *and* gate. Since both of the inputs to an *and* must be 1 for it to produce a 1, the original value of *A* must be 0, while the value of *B* must be 1. Under all other circumstances this sub-circuit produces 0. Thus, this circuit successfully captures the meaning of line two of the *xor* truth table.

A sub-circuit to implement line three of the *xor* truth table can be constructed similarly. Its Boolean expression is $Z = A \cdot \overline{B}$:

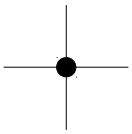


This circuit generates a 1 whenever input *A* is 1 and *B* is 0. Under all other circumstances, it produces a 0. The following figure illustrates a complete *xor* circuit, which contains the two sub-circuits joined together by an *or* gate. This is reasonable since the *xor* can be true either by way of the first sub-circuit or the second. Note that due to the manner in which the two sub-circuits were constructed, it is impossible for both of them to be true at the same time.



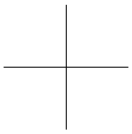
The Boolean expression for this circuit is $Z = (A \cdot \overline{B}) + (\overline{A} \cdot B)$.

A new feature introduced in this circuit diagram is the connection point. Each of the two sub-circuits making up the *xor* requires access to both inputs. So the wires that represented these inputs had to be *split* in some way. We indicate a branch (or connection) point in a circuit diagram by a dot. Connection points allow a wire to be split so that its current state can *flow* to multiple destinations. Here is what a connection point looks like graphically:



The pin on the left is the input to the connection point, or connector. The top, right, and bottom pins are the outputs. Hence, this connector splits the input wire three ways. In the *xor* circuit diagram a two-way, rather than three-way split was required, so one of the output pins is not drawn.

Connection points should not be confused with wires that just happen to cross one another by chance. In such a case there is *no* connection between the wires, so their signals do not interfere in any way. Think of the wires as insulated and just lying across one another. Wires that cross but are not connect are represented graphically in the following way:



You should convince yourself that the circuit above does indeed implement the truth table for *xor*. However, you should not come away from this discussion thinking that it is the only way (or even the most efficient way) to implement the *xor* behavior. The approach to circuit design of identifying the lines of the truth table that generate a 1, implementing sub-circuits to generate a 1 only under those circumstances, and then connecting all of the sub-circuits together via an *or*, works. But, it frequently results in circuits that are more complex than really necessary. For example, our implementation of *xor* requires five gates (not counting connectors). An implementation that requires only four gates can be developed from the Boolean expression $Z = (A + B) \cdot (\overline{A \cdot B})$.

Comparators

The purpose of a comparator is to examine two input values to determine whether a particular condition is satisfied. If the inputs satisfy the condition, the comparator generates a 1 (true). If the inputs do not satisfy the condition, the comparator generates a 0 (false).

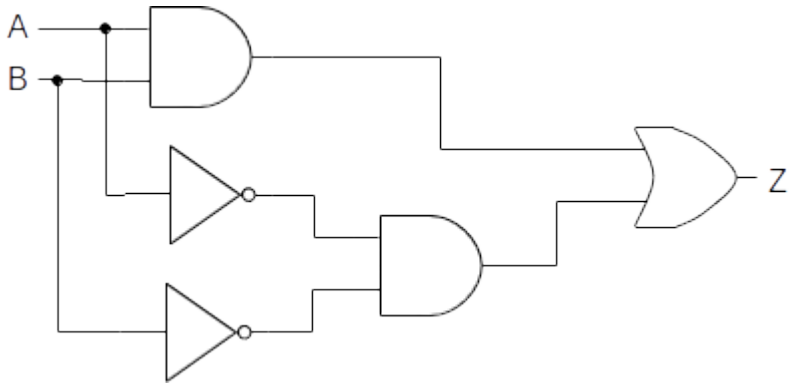
The most common type of comparator is the comparator for *equality*. This type of comparator determines whether two input values are identical. If the values are the same, the comparator generates a 1 (true). If the input values are different, the comparator generates a 0 (false).

Comparators come in different sizes, based on the width of their inputs. The simplest comparator is the one-bit comparator for equality. This circuit takes in two single-bit numbers and generates a 1 if they are equal and a 0 otherwise. Here is the truth table for the one-bit comparator for equality. The inputs are labeled *A* and *B*. The output is labeled *Z*:

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

In order to implement a circuit with this behavior, we first note the lines of the truth table that generate a 1. These are lines one and four. Let's look first at line four. This line of the table says that *Z* should be 1 when both *A* and *B* are 1. Implementing a circuit that will generate 1 under this circumstance, and no other, is trivial since the two-input *and* gate already does exactly what we want.

Producing a *sub-circuit* for line one of the table isn't really that difficult either. In order to have a sub-circuit that generates 1 when both inputs are 0, simply invert (or *not*) each of the inputs and send the results into an *and* gate. Putting these ideas together, we develop the following circuit:



Its expression is $Z = (A \cdot B) + (\overline{A} \cdot \overline{B})$.

The design for the one-bit comparator for equality can be extended to multi-bit numbers. For example, the two-bit comparator for equality has the following truth table:

A_0	A_1	B_0	B_1	Z
0	0	0	0	**1**
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	**1**
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0

A_0	A_1	B_0	B_1	Z
1	0	0	1	0
1	0	1	0	**1**
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	**1**

This table consists of 16 rows since it has four input bits. Remember, the number of rows of a truth table is always 2^n, where n is the number of input bits. In the table, A_0 represents the low-order bit of input A, and A_1 represents the high-order bit of A. Likewise, B_0 is the low-order bit of B and B_1 is the high-order bit. Hence, the first row of the table represents inputs of A=00 and B=00, which are equal; so the output, Z, is 1 (true). Likewise, the second row of the table represents A=00 and B=01, which are not equal; so the output, Z, is 0 (false).

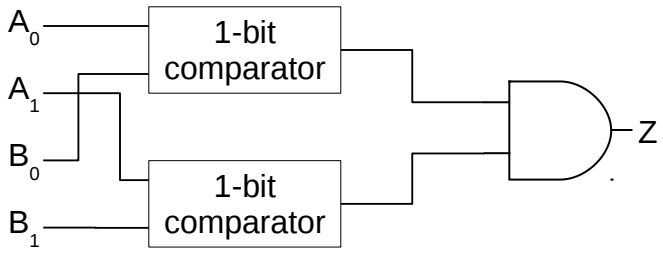
To get a clearer idea of what this circuit does, let's examine only those rows of the table in which the output is 1 (true). There are four cases in which the two-bit comparator for equality generates a 1:

A_0	A_1	B_0	B_1	Z
0	0	0	0	**1**
0	1	0	1	**1**
1	0	1	0	**1**
1	1	1	1	**1**

The first row corresponds to the case where both A and B are zero (00). The second row captures the case where both inputs are one (01), the third where the inputs are two (10), and the fourth where they are three (11). Note that these are binary inputs, and the fact that, for example, 11 is three deals with the representation of binary numbers and their decimal equivalents. This is something that will be covered in detail later.

How can we build a circuit with this behavior? In the previous examples we built sub-circuits to handle each case in which the main circuit was to produce 1. We then connected the sub-circuits together using an *or* gate. Because the present circuit is a multi-bit version of a circuit we have already constructed, we will take a different approach to the circuit design problem. When attempting to construct a multi-bit version of a single-bit circuit, it is best to approach the problem by looking for ways in which single-bit versions of the circuit can be interconnected to form the multi-bit circuit.

A careful inspection of the above table will show that a 1 should be generated whenever the low-order bits of both inputs (A_0 and B_0) are equal *and* the high-order bits of the inputs (A_1 and B_1) are equal as well. Hence, routing inputs A_0 and B_0 into a one-bit comparator for equality and inputs A_1 and B_1 into a separate one-bit comparator for equality, then sending both of these results into an *and* gate, will produce a circuit with the desired behavior. Here is the circuit:



The circuit is based on the idea that two-bit numbers are equal if both their low-order and high-order bits are identical. An actual implementation of the circuit would require that the boxes marked "one-bit comparator" be replaced with comparator circuitry along the lines of that shown earlier.

In addition to comparators for equality, comparators for other conditions (such as *less than* and *greater than*) can be constructed. While most of these are not covered in this lesson, here's a brief example of the truth table for the one-bit comparator for *less than* (i.e., $A < B$):

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

Can you come up with the circuit for this logic in the space below?

Can you complete the truth table for the *greater than* comparator (i.e., $A > B$)?

A	B	Z

Can you come up with the circuit for this logic in the space below? Here's an example circuit:

Searching

One of the most common tasks that is undertaken in Computer Science is the task of searching. We can search for many things: a value (like a phone number), a number (like seven), a position (like who finished the race in second place), or an object (like an image of a dog). Typically, searching will require doing a specific task over and over again (i.e., searching in one position, and then searching in another position), until we find what we are looking for. This requires repetition. We have already learned how to deal with and represent repetition. Our first task is to design an algorithm that can be used to search for a given value.

Activity 1

For this activity, you will participate in the demonstration of an algorithm to find some value (specifically, the maximum value) in a list of values. Approximately ten students will be asked to stand in front of the class. These will be known as *memory* students; that is, they represent a subset of a computer's memory. Each *memory* student will secretly write a number of their choosing on a post-it note. Collectively, this represents a list of numbers stored in the computer's memory.

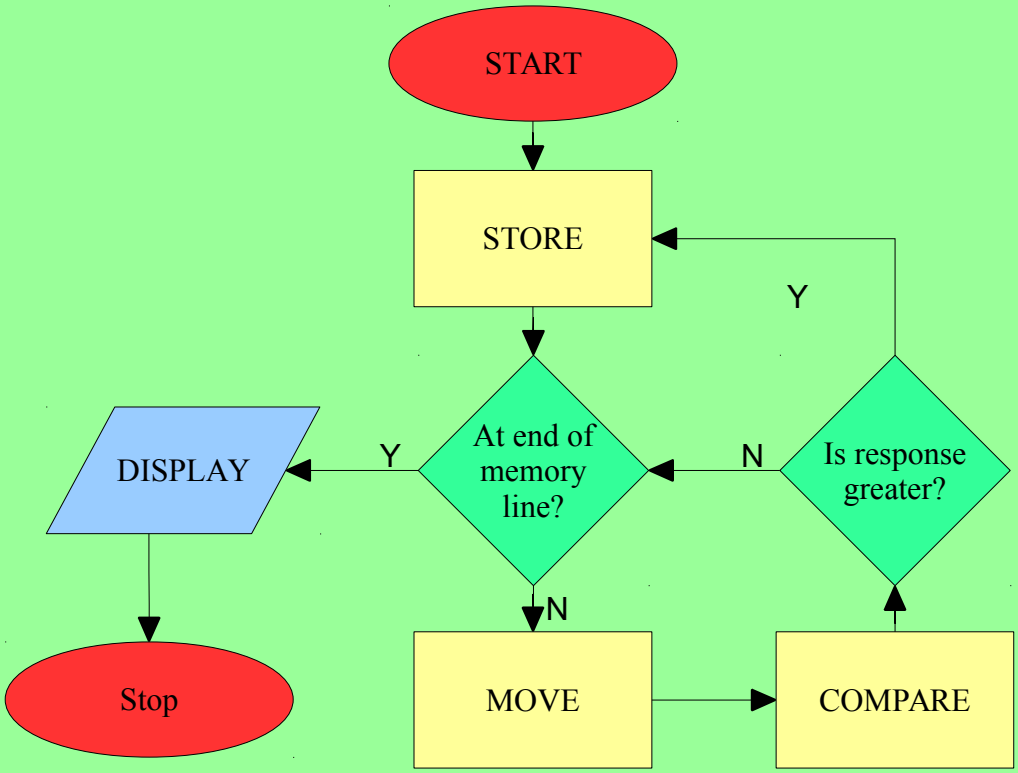
One other student, known as the *computer* student, will represent the computer as it executes the algorithm. This student will follow instructions (like a computer would) and keep track of a single value representing the current maximum value in the list of numbers. This student will have a few post-it notes (or a small dry-erase board) to keep up with the current maximum value. The *computer* student understands several instructions:

START – instructs the *computer* student to go to the beginning of the *memory* student line;
MOVE – instructs the *computer* student to move to the next *memory* student in the line;
COMPARE – instructs the *computer* student to compare the number of the current *memory* student to the one currently recorded and respond with *greater*, if the current *memory* student's number is greater, or *less*, otherwise;
STORE – instructs the *computer* student to record the current *memory* student's number; and
DISPLAY – instructs the *computer* student to display the currently recorded value.

The suggested algorithm is described in pseudocode as follows:

```
START
STORE
repeat
      MOVE
      COMPARE
      if the response is greater
      then
            STORE
      end
until the computer student is at the end of the memory student line
DISPLAY
```

The suggested algorithm is described using a flowchart as follows:



Note that the algorithm initially instructs the *computer* student to record the first number in the list and assume it to be the maximum value. Technically, it is the maximum value *at that time*, but it provides a starting point. The algorithm then proceeds to compare each of the other values in the list and replaces the recorded maximum value when a greater value is encountered in the list.

At the end of the algorithm, the *computer* student should have the maximum value recorded on the post-it note, which is revealed to the class. For effect, the *memory* students should subsequently display their post-it notes to confirm the maximum value.

Note that the algorithm works regardless of the arrangement of the *memory* students.

Sequential search

The activity above was a demonstration of the sequential search. Sequential searching has a lot of applications in computer science.

Definition: ***Sequential search*** *(also known as* ***linear search****) is the process of locating a value in a list of values by checking each element, one at a time, until either the value is located or the end of the list has been reached.*

The algorithm described in the activity above can be represented more generally in pseudocode as follows:

```
1: i ← 1
2: max ← value of item i in the list
```

```
 3: repeat
 4:     increment i
 5:     if value of item i in the list > max
 6:     then
 7:         max ← value of item i in the list
 8:     end
 9: until i = the length of the list
10: display max
```

This algorithm searches for (and displays) the largest value in a list of numbers. How could it be modified to search for (and display) the smallest value in a list of numbers? Try to modify the algorithm above in the space below:

```
 1:
 2:
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
```

The sequential search can also be used to find a specific value (such as 100) instead of something more general as we did above (such as the maximum value). How could the algorithm that finds the maximum value be modified to search for a specified value and display whether it was found (or not)? Try to modify the algorithm above in the space below:

```
 1:
 2:
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
11:
```

Is this algorithm efficient? What happens if the value searched for is not in the list? What happens if the value searched for is near the beginning of the list? We observe that, if the value searched for is near the beginning of the list, the algorithm sets the variable *found* to `true` but continues to look through the remainder of the list. This is not very efficient. Consider a list of 1 million values such that the value searched for happens to be found at the beginning of the list. The algorithm above would absolutely find

the value at the beginning of the list; however, it would then continue to needlessly search through the remaining 999,999 values in the list. There is no stop condition in the case that the value searched for is found. We can fix this quite easily, however, by slightly modifying the algorithm:

```
 1: n ← value to search for
 2: i ← 1
 3: found ← false
 4: repeat
 5:     if value of item i in the list = n
 6:     then
 7:         found ← true
 8:     end
 9:     increment i
10: until i > the length of the list or found = true
11: display found
```

By adding an additional **exit condition** (something that terminates the repetition), we can capture the case that the value is found at any point in the list and immediately terminate the search. When designing algorithms, we must carefully examine them to find inconsistencies, problems, and to ensure that they, in fact, solve the problem. In addition, we must not forget to consider efficiency, performance, and scalability. For a very short list, efficiency may not matter; however, we don't usually deal with small amounts of data in practice.

Here is an example of the sequential search for the value 80 applied to a list containing the following values: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100:

Search List	Comparison	Action
10, 20, 30, 40, 50, 60, 70, 80, 90, 100	$10 \neq 80$	Target not found; continue
10, **20**, 30, 40, 50, 60, 70, 80, 90, 100	$20 \neq 80$	Target not found; continue
10, 20, **30**, 40, 50, 60, 70, 80, 90, 100	$30 \neq 80$	Target not found; continue
10, 20, 30, **40**, 50, 60, 70, 80, 90, 100	$40 \neq 80$	Target not found; continue
10, 20, 30, 40, **50**, 60, 70, 80, 90, 100	$50 \neq 80$	Target not found; continue
10, 20, 30, 40, 50, **60**, 70, 80, 90, 100	$60 \neq 80$	Target not found; continue
10, 20, 30, 40, 50, 60, **70**, 80, 90, 100	$70 \neq 80$	Target not found; continue
10, 20, 30, 40, 50, 60, 70, **80**, 90, 100	$80 = 80$	Target found; stop

The search begins by comparing the target value, 80, with the first value in the list, 10. Since $10 \neq 80$, then the search continues to the next value in the list. This is continued until the value is found in the eighth position. This sequential search requires a total of eight searches to find the target value. We can apply the sequential search to a number guessing problem: on average, how many tries would it take to correctly guess a number from one to 1,000? In the best case, one. In the worst case, 1,000. On average, 500.

Let's generalize this for a list of *n* values and call the tries *comparisons* (since that's what is actually being done). In the best case, it takes one comparison (if the target value is at the beginning of the list). In the worst case, it takes *n* comparisons (if the target value is at the end of the list). If many searches for various target values were performed, an *average* number of comparisons could be calculated – and

we would find that approximately half of the list would need to be searched through. This should not be surprising, since the *good* cases (where the target value is at the front of the list), and the *bad* cases (where the target value is at the end of the list) tend to cancel each other out over many searches. On average, searching for a target value in a list containing *n* items requires *n* / 2 comparisons.

Activity 2

Can you come up with a better algorithm to correctly guess a randomly picked number from one to 1,000? Students will be paired up in groups of two. One student (called the *picker*) will secretly pick a number from one to 1,000; the other student (called the *guesser*) will attempt to guess the secret number under the following constraints:

The *guesser* submits a single number at a time;
The *picker* replies **HIGHER** if the secret number is higher than the *guesser*'s submission, or **LOWER** otherwise; and
The *picker* replies **CORRECT** if the *guesser*'s submission is correct.

Each group of students should design an algorithm to correctly **guess** the number picked by the *picker*. Record the number of guesses that it took to guess correctly. Then switch places: the *picker* becomes the *guesser* and vice versa.

Binary search

Some of you may have realized that the runtime of a sequential search is dependent on the size of the list that is being searched. Sequentially searching through a list of 1,000 items (as in the activity above) will typically take ten times as long as searching through a list of 100 items (i.e., 100 is one-tenth of 1,000). In the best case, the item will be found at the beginning of the list (i.e., it requires searching through only one item). In the worst case, the item will be found at the end of the list or not found at all (i.e., it requires searching through the entire list). On average, the algorithm would have to search through about half of the list.

Consider the scenario of searching for a name that corresponds to a specific phone number in a phone book. This is the reverse of what is normally done (i.e., searching for a phone number that corresponds to a given name). How could this be done? The only way is to perform a sequential search, starting at the beginning of the phone book. This could take a long time, and it would be utterly depressing if the phone number was not found in the phone book. While the sequential search is effective, there are scenarios (like this one) where a more efficient way of searching is preferable.

Consider the normal approach to searching a phone book for a phone number that corresponds to a given name. A sequential search would require starting the search at the beginning of the phone book and continue until either the name is found or until the entire phone book is exhausted. However, a phone book has a specific quality that can be taken advantage of which makes searching significantly easier and faster: it is ordered in a meaningful way. The names in a phone book are arranged in alphabetical order. This makes searching easier in that only a small subset of the names have to actually be searched through. When searching through a phone book, the usual process is to thumb through it until the first letter of the given name is reached. From there, a sequential search (of sorts) is performed to search for the given name.

A method that may be better suited for computing is to open the phone book in the middle. If the given name starts with a letter in the alphabet that comes before the one that is there, then the right half of the phone book can be ignored. With a single comparison, half of the phone book has been eliminated.

This strategy can be continued by shifting to the part of the phone book that represents the halfway point in *the first half* of the phone book. Another comparison is performed to determine if the given name appears before or after this point. This continues until the given name is found. With each check, half of the remaining portion is eliminated.

This strategy can be used to guess a number from one to 1,000 much more efficiently than by doing a sequential search. So how is the halfway point (or middle value) calculated? Given a list of n items, we calculate the middle as follows:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1$$

Note that the brackets represent the *floor* function which means to round down to the largest previous integer. For example, the floor of 3.14 is 3, and the floor of 27.9 is 27. For an odd number of values, this selects the middle value (i.e., there are an equal number of values on either side of the selected value). For an even number of values, this selects the value to the right of the "middle" (i.e., there is one more value to the left of the selected value than there are to the right). The following simpler formula may, at first, seem equivalent:

$$\left\lceil \frac{n}{2} \right\rceil$$

Indeed, it is for odd values of n; e.g., for n=5:

$$\left\lfloor \frac{5}{2} \right\rfloor + 1 = \lfloor 2.5 \rfloor + 1 = 2 + 1 = 3$$

$$\left\lceil \frac{5}{2} \right\rceil = \lceil 2.5 \rceil = 3$$

However, it is not equivalent for even values of n; e.g., for n=6:

$$\left\lfloor \frac{6}{2} \right\rfloor + 1 = \lfloor 3 \rfloor + 1 = 3 + 1 = 4$$

$$\left\lceil \frac{6}{2} \right\rceil = \lceil 3 \rceil = 3$$

Some of you may have naturally implemented this algorithm in the activity above. Let's formally define it now in pseudocode:

```
1: repeat
2:     n ← number of items in the current portion of the list
3:     mid ← floor(n / 2) + 1
4:     guess mid
5:     if response is HIGHER
6:     then
7:         discard the left half of the list
8:     else if response is LOWER
```

```
 9:      then
10:          discard the right half of the list
11:      end
12: until guess is correct
```

This algorithm is known as a binary search. Note that it supposes that every number from 1 to *n* is in the list as in the activity above.

Definition: ***Binary search*** *is the process of locating a value in an* ***ordered*** *list of values by repeatedly comparing the value in the middle of the relevant portion of the list to the desired value and discarding the appropriate half of the list. Searching terminates when either the desired value is located or the list can no longer be divided in half.*

Here is an example of the binary search for the value 70 applied to a list containing the following values: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100:

Search List	Comparison	Action
10, 20, 30, 40, 50, **60**, 70, 80, 90, 100	60 < 70	Target not found; discard left half
70, 80, **90**, 100	90 > 70	Target not found; discard right half
70, **80**	80 > 70	Target not found; discard right half
70	70 = 70	Target found; stop

The list is initially split in the middle at 60 (since there are 10 values, and floor(10/2) + 1 is 6 – the sixth value in the list). Since 60 is less than 70, we can safely discard the left half of the list (including the split value 60) and continue with the right half of the list: 70, 80, 90, 100. This smaller sub-list is split in the middle at 90. Since 90 is greater than 70, we can safely discard the right half of the sub-list (including the split value 90) and continue with the left half of the sub-list: 70, 80. This smaller sub-list is split in the middle at 80. Since 80 is greater than 70, we can safely discard the right half of the sub-list (including the split value 80) and continue with the left half of the sub-list: 70. This sub-list has a single value (70) which, when compared to 70, is found to be the target value. In four comparisons, the target value was found in the list.

What would happen if we tried to search the list for a value that the list doesn't contain? Here is an example of the binary search for the value 45 applied to a list containing the same values as before:

Search List	Comparison	Action
10, 20, 30, 40, 50, **60**, 70, 80, 90, 100	60 > 45	Target not found; discard right half
10, 20, **30**, 40, 50	30 < 45	Target not found; discard left half
40, **50**	50 > 45	Target not found; discard right half
40	40 < 45	Target not found; discard left half
empty list	none	Target not found; stop

It should be evident that the binary search is significantly faster than a sequential search. It turns out that guessing a number from 1 to 10,000 using the binary search will take, at most, 14 guesses.

Intuitively, this is because 10,000 can be divided in half roughly 14 times: 10,000 is reduced to 5,000, then to 2,500, then to 1,250, then to 625, then to 313, then to 157, then to 79, then to 40, then to 20, then to 10, then to 5, then to 3, then to 2, and finally to 1 (14 total splits). We can actually calculate this precisely by solving the following equation:

$$2^n = 10000$$

Exponentiation is the reverse of logarithms. That is, $2^n = 10000$ expresses the same relationship as $\log_2 10000 = n$. We can visualize how the binary search works to find a value in a list of 10,000 values by illustrating each guess (the middle value):

Comparison	List size	Items on left	Middle value	Items on right	Remaining items
1	10,000	5,000	5,001	4,999	5,000
2	5,000	2,500	2,501	2,499	2,500
3	2,500	1,250	1,251	1,249	1,250
4	1,250	625	626	624	625
5	625	312	313	312	312
6	312	156	157	155	156
7	156	78	79	77	78
8	78	39	40	38	39
9	39	19	20	19	19
10	19	9	10	9	9
11	9	4	5	4	4
12	4	2	3	1	2
13	2	1	2	0	1
14	1	0	1	0	0

Consider a simpler problem of guessing a number from 1 to 1,000. How many guesses would that take? We know that $2^{10} = 1024$ and that $2^9 = 512$; therefore, 1,000 can be divided by two between 9 and 10 times. However, it's evident that it's closer to 10 than it is to 9. In fact, it is actually 9.97 times. Recall that, on average, it would take 500 guesses if the sequential search were used instead. The binary search is therefore 50 times faster than the sequential search for a list of 1,000 values (500 guesses / 10 guesses = 50). What about the comparison for a list of 1 billion values? The sequential search would take, on average, 500 million comparisons. The binary search would take, at worst, 30 comparisons. That's almost 17 million times faster!

Did you know?

To solve for n in $2^n = 1000$, we can use the formula that expresses the inverse of raising two to a power: $\log_2 1000 = n$. Logarithms represent the power to which some number, called the base (in this case, 2), must be raised to produce a given number (in this case, 1,000). Most calculators do not have a $\log_2$ function; however, they typically do have a $\log_{10}$ function (note that most calculators denote this as *log*

and omit the base). We can convert easily by using the following conversion (where *b* is the given base and *d* is the desired base):

$$\log_b x = \frac{\log_d x}{\log_d b}$$

In the example above where the given base is 2, the target base is 10, and *x* is 1000, we can convert as follows:

$$\log_2 1000 = \frac{\log_{10} 1000}{\log_{10} 2} = 9.97$$

For giggles, how many tries would it take to guess a number from one to 1 billion?

$$\log_2 1\,billion = \frac{\log_{10} 1\,billion}{\log_{10} 2} = 29.9$$

Searching for a target value in a list containing *n* items requires, *at maximum*, the following number of comparisons:

$$\lceil \log_2 (n+1) \rceil$$

Note that the brackets represent the *ceiling* function which means to round up to the smallest following integer. For example, the ceiling of 3.14 is 4, and the ceiling of 27.1 is 28. You may be confused why we're taking the logarithm of *n*+1. Consider the simplest case of a list containing a single item (i.e., *n* = 1). $\lceil \log_2 (1+1) \rceil = 1$. Clearly, a list containing a single value requires at most a single comparison!

The following table shows the number of comparisons required to search through a list of values ranging from 0 to 10,000 items using both the sequential search (average number of comparisons) and binary search (maximum number of comparisons). It also includes a performance measure that compares how much better the binary search is when compared to the sequential search. It is amazing to see the huge difference in the performance of the two algorithms:

Number of items	Sequential search	Binary search	Performance
0	0	0	0
1,000	500	10	50
2,000	1,000	11	91
3,000	1,500	12	125
4,000	2,000	12	167
5,000	2,500	13	192
6,000	3,000	13	231
7,000	3,500	13	269

Number of items	Sequential search	Binary search	Performance
8,000	4,000	13	308
9,000	4,500	14	321
10,000	5,000	14	357

For a list of 1,000 items, the binary search is roughly 50 times faster than the sequential search, and for a list of 10,000 items, the binary search is roughly 357 times faster. This, however, does not take into consideration that the binary search takes extra calculations (i.e., calculating the middle of the current portion of the list, discarding half of the list, etc). However, suppose that each binary search comparison takes 1/10 of a second and each sequential search comparison takes 1/1000 of a second. We can still observe that the binary search ridiculously outperforms the sequential search as illustrated in Figure 1.

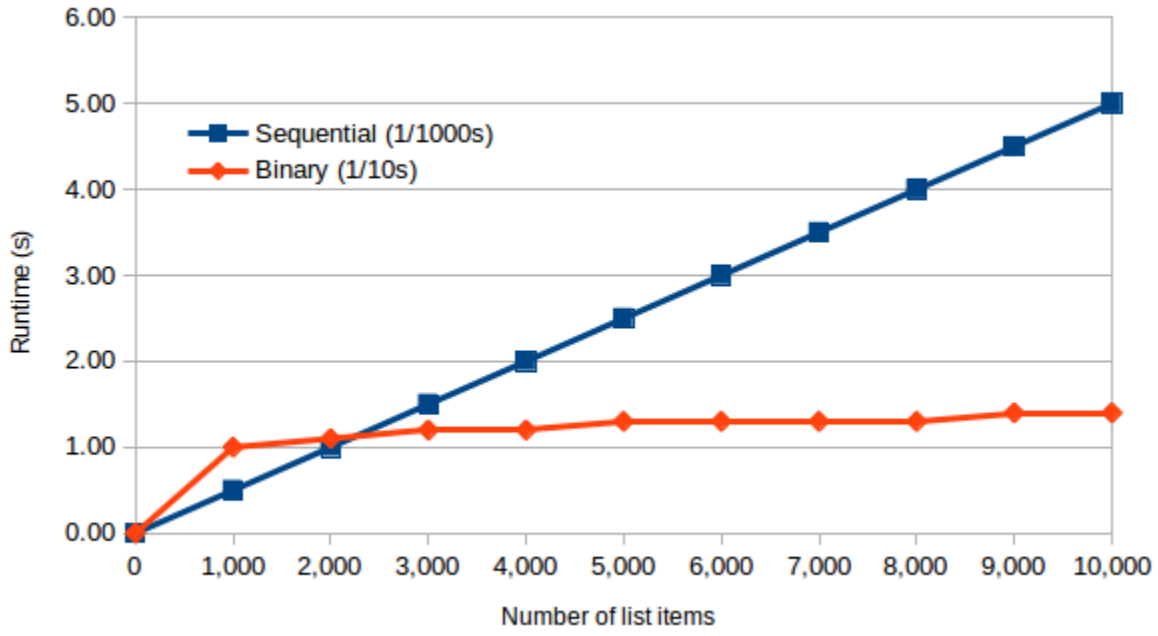


Figure 1: Runtime comparison of sequential and binary search

Also note that the time it takes to initially order the list of numbers is not considered. Since the binary search requires the list to be ordered, it is worthwhile to investigate various ordering methods and their performance.

Sorting

We have seen that the binary search is significantly faster than the sequential search. Why then do we even need to know about the sequential search? Why not use the binary search every time we need to search for something? What weaknesses does the binary search have? What scenarios can you imagine where a sequential search would be preferable to a binary search? The answer lies in the fact that a binary search can only be used when the data to be searched through is ordered in some meaningful way. This does not occur naturally (i.e., we must order the data first, prior to searching).

Humans seem to have a basic desire to structure and organize the world around them. One of the most basic ways of organizing a collection of objects is to arrange them according to some common characteristic such as size, weight, cost, or age. A group of objects is **sorted** when the objects are arranged according to some rule that involves the use of *orderable* **keys**. While the term *orderable* will not be rigorously defined, intuitively it means keys on which the concepts of *less than*, *greater than*, and *equal to* make sense, or on which *precedes* and *follows* have meaning. Orderable keys include numbers such as weight, height, age, income, and social security number. Strings of characters such as words, names, and addresses are also orderable. Numeric keys are generally arranged from smallest to largest, or, occasionally, largest to smallest.

Sorting is a very important problem. If information is not organized in some fashion, retrieval can be quite time consuming. To find a particular item in an unordered (or unsorted) list, we are forced to use the sequential search rather than the more efficient binary search. While the sequential search approach may be acceptable for a small numbers of items, it becomes impractical when dealing with large numbers of objects. For this reason, computer scientists have focused a great deal of attention on the sorting problem and have devised a number of sorting algorithms. Over the years, these algorithms have been closely studied and the efficiency of each carefully analyzed.

This section presents three common approaches to the sorting problem: the **bubble sort**, the **selection sort**, and the **insertion sort**. As was the case with sequential and binary searches, each of the three sorts will be compared to determine their efficiency. In order to develop estimates of the runtimes of the three sort algorithms, we will use a computer capable of executing one million comparisons per second. While this may seem like a lot and to be very fast, today's computers are many times more powerful. In fact, they are capable of performing billions of basic low-level instructions per second. The number of higher-level operations they can perform each second depends on many factors other than CPU speed. In the case of sorting, the number of comparisons per second depends on factors such as whether the list being sorted is in main memory or on disk, and whether the computer is running other programs besides the sort procedure.

The take home message is not to place too much emphasis on the numbers that will be generated for predicted runtimes as they are highly dependent on the assumptions made concerning computing hardware. Instead, what is important is the *relative* performance of the algorithms, which is not affected by underlying hardware assumptions.

Bubble sort

The bubble sort processes a list of items from left-to-right. The sorted list is built, *in place*, from right-to-left; that is, the largest value in the list is placed in its final position first, and so on, in the same list. It works by repeatedly comparing two neighboring elements (i.e., in positions *i* and *i*+1). If the two neighboring elements are out of order, they are swapped. The algorithm advances the position *i* and repeats. If, at the end of one iteration of the entire list (also called a **pass**) any swaps were made, the process is repeated. If no swaps were made, it means the list is sorted, and the process terminates. The following is an example of the bubble sort on the list [7 9 3 5 1], with the underlined elements showing where the comparisons are made in each pass and red indicating the sorted portion of the list:

Pass 1		
List	**Comparison**	**Action**
7 9 3 5 1	7 < 9	no swap

7 9 3 5 1	9 > 3	swap
7 3 9 5 1	9 > 5	swap
7 3 5 9 1	9 > 1	swap
7 3 5 1 9	done with this pass	
Pass 2		
List	**Comparison**	**Action**
7 3 5 1 9	7 > 3	swap
3 7 5 1 9	7 > 5	swap
3 5 7 1 9	7 > 1	swap
3 5 1 7 9	done with this pass	
Pass 3		
List	**Comparison**	**Action**
3 5 1 7 9	3 < 5	no swap
3 5 1 7 9	5 > 1	swap
3 1 5 7 9	done with this pass	
Pass 4		
List	**Comparison**	**Action**
3 1 5 7 9	3 > 1	swap
1 3 5 7 9	done with the sort	

This sort is called the bubble sort because, during each pass, larger values *bubble* to the end of the list. Note that the bubble sort required four passes through the list of items. That is, it needed to go through some portion of the list starting at the beginning, compare values, and potentially swap values four times. In general for a list of *n* items, the bubble sort requires *n*-1 passes through the list. Did you notice that, as each pass was made, fewer and fewer elements in the list were compared. In the first pass, for example, four comparisons were made; in the second pass, three comparisons were made; in the third pass, two comparisons were made; and in the final pass, only one comparison was made. If we consider all passes in aggregate, on average about half of the values were compared; therefore, for a list of *n* items, an average of *n* /2 comparisons are made at each pass.

We can now calculate the number of comparisons made using the bubble sort as follows:

*comparisons = number of passes * average number of comparisons per pass*

As stated, for a list of *n* items, *n*-1 passes are required, each of which *on average* makes *n*/2 comparisons. In total, the number of comparisons made using the bubble sort is then:

$$(n-1)*\frac{n}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$$

For the list above of five items, the number of comparisons is then:

$$\frac{1}{2}(5^2-5) = \frac{1}{2}(20) = 10$$

In each pass of the bubble sort on the list in the example above, swaps were made. Let's take a look at an example where the list of items is almost already sorted:

Pass 1		
List	**Comparison**	**Action**
1 2 3 5 4	1 < 2	no swap
1 2 3 5 4	2 < 3	no swap
1 2 3 5 4	3 < 5	no swap
1 2 3 5 4	5 > 4	swap
1 2 3 4 5	done with this pass	
Pass 2		
List	**Comparison**	**Action**
1 2 3 4 5	1 < 2	no swap
1 2 3 4 5	2 < 3	no swap
1 2 3 4 5	3 < 4	no swap
1 2 3 4 5	done with the sort	

Since no swaps were made in pass 2, then the sort is immediately terminated. Terminating the bubble sort during a pass if no swaps are made is a *tweak* on the original bubble sort called the optimized bubble sort.

The method illustrated on the sort the list [7 9 3 5 1] in the table above is great for showing each comparison and illustrating the bubble sort in detail; however, it is overkill when simply showing how a list is sorted. We can slightly deviate from the method used previously and instead summarize each pass in a single row of the table. The underlined items in the list at each pass represent the unsorted portion of the list:

Pass	**List**
original list	7 9 3 5 1
1	7 3 5 1 9
2	3 5 1 7 9
3	3 1 5 7 9
4	1 3 5 7 9

Show how the bubble sort works on the list [9 1 2 5 3 7] by completing the table below:

Pass	List
original list	9 1 2 5 3 7

Did you know?

Note the **for-next** construct in the pseudocode for the bubble sort above. It is used to repeat a *fixed* or *known* number of times. You have already seen the **repeat-until** construct that repeats until some condition is true. A *for-next* construct can be rewritten into a *repeat-until* construct as follows:

The *for-next* version:

```
for i ← 1..n
        ...
next
```

The *repeat-until* version:

```
i ← 1
repeat
        ...
        i ← i + 1
until i = n
```

It is not always possible to rewrite a repeat-until construct into a for-next construct; for example (assuming that the length of the list is unknown):

```
repeat
        remove item from list
until the list is empty
```

In this case, there is no way of knowing when the list will be empty if we don't originally know the length of the list (as assumed).

Selection sort

The selection sort is probably the most natural approach to sorting. Most people would likely come up with this algorithm when asked to design one that, for example, reorders a line of people based on their height. The selection sort processes a list of items from left-to-right. The sorted list is also built, in place, from left-to-right; that is, the smallest value in the list is placed in its final position first, and so on. Since the list is sorted in place, we keep track of both an unsorted portion and a sorted portion. And since the list is built from left-to-right, then the sorted portion is on the left. The selection sort works by repeatedly finding the smallest value in the unsorted portion of the list and swapping it with the first value in the unsorted portion. This action increases the sorted portion of the list and shrinks the unsorted

portion of the list by one value. This is repeated until there are no values in the unsorted portion of the list.

The following is an example of the selection sort on the list [7 9 3 5 1] with the underlined elements showing where the comparisons are made in each pass and red indicating the sorted portion of the list:

Pass 1			
List	**Comparison**	**Smallest so far**	**Action**
7 9 3 5 1	7 < 9	7	
7 9 3 5 1	7 > 3	3	
7 9 3 5 1	3 < 5	3	
7 9 3 5 1	3 > 1	1	
1 9 3 5 7	done with this pass		swap 7 and 1
Pass 2			
List	**Comparison**	**Smallest so far**	**Action**
1 9 3 5 7	9 > 3	3	
1 9 3 5 7	3 < 5	3	
1 9 3 5 7	3 < 7	3	
1 3 9 5 7	done with this pass		swap 9 and 3
Pass 3			
List	**Comparison**	**Smallest so far**	**Action**
1 3 9 5 7	9 > 5	5	
1 3 9 5 7	5 < 7	5	
1 3 5 9 7	done with this pass		swap 9 and 5
Pass 4			
List	**Comparison**	**Smallest so far**	**Action**
1 3 5 9 7	9 > 7	7	
1 3 5 7 9	done with the sort		swap 9 and 7

Note that the selection sort also required four passes through the list of items. For a list of *n* items, the selection sort requires *n*-1 passes through the list. Again, note that as each pass was made, fewer and fewer elements in the list were compared. If we consider all passes in aggregate, on average about half of the values were compared; therefore, for a list of *n* items, an average of *n* /2 comparisons are made at each pass.

We can now calculate the number of comparisons made by using the formula defined above:

*comparisons = number of passes * average number of comparisons per pass*

For a list of *n* items, *n*-1 passes are required, each of which *on average* makes *n*/2 comparisons. In total, the runtime performance of the selection sort is then:

$$(n-1)*\frac{n}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$$

For the list above of five items, the number of comparisons is then:

$$\frac{1}{2}(5^2-5) = \frac{1}{2}(20) = 10$$

Again, we can show how the selection sort works by summarizing each pass in a single row of the table. The underlined items in the list at each pass represent the unsorted portion of the list:

Pass	List
original list	7 9 3 5 1
1	1 9 3 5 7
2	1 3 9 5 7
3	1 3 5 9 7
4	1 3 5 7 9

This is sufficient to show how the selection sort works. Can you design an algorithm in pseudocode for the selection sort? Try it in the space below:

```
 1:
 2:
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
11:
```

Show how the selection sort works on the list [9 1 2 5 3 7] by completing the table below:

Pass	List
original list	9 1 2 5 3 7

Insertion sort

The insertion sort is the procedure that most people use to arrange a hand of cards. To begin to understand the algorithm, think of the list as having both a sorted portion and an unsorted portion (as we did in the previous sorts). The first item of the list is considered to be a sorted list one item long, with the rest of the list (items 2 through *n*) forming an unsorted portion.

The insertion sort removes the *first* item from the unsorted portion of the list and marks it as the item to be inserted. It then works its way from the *back* to the *front* of the sorted portion of the list, at each step comparing the item to be inserted with the current item. As long as the current item is larger than the item to be inserted, the algorithm continues moving *backward* through the sorted portion of the list. Eventually it will either reach the beginning of the sorted portion of the list or encounter an item that is less than or equal to the item to be inserted. When that happens the algorithm inserts the item at the current insertion point.

The entire process of selecting the first item from the unsorted portion of the list and scanning backwards through the sorted portion of the list for the insertion point is then repeated. Eventually, the unsorted portion of the list will be empty since all of the items will have been inserted into the sorted portion of the list. When this occurs, the sort is complete.

The following is an example of the insertion sort on the list [7 9 3 5 1] with the underlined elements showing where the comparisons are made in each pass, the bold value indicating the current *first* item in the unsorted portion of the list, and red indicating the sorted portion of the list:

Pass 1			
List	**First item**	**Comparison**	**Action**
7 3 5 1	9	7 < 9	
7 9 3 5 1		done with this pass	insert 9
Pass 2			
List	**First item**	**Comparison**	**Action**
7 9 5 1	3	9 > 3	slide 9 over
7 9 5 1	3	7 > 3	slide 7 over
3 7 9 5 1		done with this pass	insert 3

Pass 3			
List	**First item**	**Comparison**	**Action**
3 7 9 1	5	9 > 5	slide 9 over
3 7 9 1	5	7 > 5	slide 7 over
3 7 9 1	5	3 < 5	
3 5 7 9 1		done with this pass	insert 5
Pass 4			
List	**First item**	**Comparison**	**Action**
3 5 7 9	1	9 > 1	slide 9 over
3 5 7 9	1	7 > 1	slide 7 over
3 5 7 9	1	5 > 1	slide 5 over
3 5 7 9	1	3 > 1	slide 3 over
1 3 5 7 9		done with the sort	insert 1

Again, we can show how the insertion sort works by summarizing each pass in a single row of the table. The underlined items in the list at each pass represent the unsorted portion of the list:

Pass	List
original list	7 9 3 5 1
1	7 9 3 5 1
2	3 7 9 5 1
3	3 5 7 9 1
4	1 3 5 7 9

Show how the insertion sort works on the list [9 1 2 5 3 7] by completing the table below:

Pass	List
original list	9 1 2 5 3 7

Note that the insertion sort required four passes through the list of items. For a list of *n* items, the insertion sort requires *n*-1 passes through the list, which is the same as both the bubble sort and the selection sort. However, the number of comparisons made at each pass is not so simple to calculate because it depends on the original state of the list. Sometimes, the *first item* in the unsorted portion of

the list will be greater than everything in the sorted portion of the list, and therefore only require one comparison (to the last value in the sorted portion of the list). At other times, it may actually be the smallest value in the entire list and thus be compared to every value in the sorted portion of the list.

Since the number of comparisons made in the insertion sort is not so straightforward, we will need to determine formulas that represent the number of comparisons for the worst case, best case, and average case.

In the best case, only a single comparison will be made at each pass (i.e., the first item in the unsorted portion of the list is in its proper place in the list). This will happen, for example, if the list is already sorted. For a list of *n* items, the insertion sort requires *n*-1 passes, and for each pass, one comparison is made in the best case. The total number of comparisons in the best case is then:

$$(n-1)*1 \;=\; n-1$$

For the list above of five items, the number of comparisons in the best case is then four (5 – 1 = 4). Here is an example on the already sorted list [1 2 3 4 5]:

Pass 1			
List	**First item**	**Comparison**	**Action**
1 3 4 5	2	1 < 2	
1 2 3 4 5		done with this pass	insert 2
Pass 2			
List	**First item**	**Comparison**	**Action**
1 2 4 5	3	2 < 3	
1 2 3 4 5		done with this pass	insert 3
Pass 3			
List	**First item**	**Comparison**	**Action**
1 2 3 5	4	3 < 4	
1 2 3 4 5		done with this pass	insert 4
Pass 4			
List	**First item**	**Comparison**	**Action**
1 2 3 4	5	4 < 5	
1 2 3 4 5		done with the sort	insert 5

In the worst case, the first item belongs at the beginning of the sorted portion of the list and therefore is compared to every value in the sorted portion of the list. In the first pass, only one comparison is made; in the second pass, two comparisons are made; and so on. On average, then, *n*/2 comparisons are made per pass. For a list of *n* items, the insertion sort requires *n*-1 passes, and for each pass, an average of *n*/2 comparisons are made in the worst case. The total number of comparisons in the worst case is then:

$$(n-1)*\frac{n}{2} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n)$$

For a list of five items, the number of comparisons in the worst case is then:

$$\frac{1}{2}(5^2-5) = \frac{1}{2}(20) = 10$$

In the first example shown with the list [7 9 3 5 1], the worst case indeed occurred (and 10 comparisons were made). Note that the list is nearly reverse sorted!

For the average case, we simply need to calculate the average **of the average** number of comparisons made in the best and worst cases. In the best case, only one comparison is made per pass; in the worst case, *n*/2 comparisons are made. The average is then *n*/4 comparisons. For a list of *n* items, the insertion sort requires *n*-1 passes, and for each pass, an average of *n*/4 comparisons are made in the average case. The total number of comparisons in the average case is then:

$$(n-1)*\frac{n}{4} = \frac{1}{4}n(n-1) = \frac{1}{4}(n^2-n)$$

For a list of five items, the number of comparisons in the average case is then:

$$\frac{1}{4}(5^2-5) = \frac{1}{4}(20) = 5$$

To summarize, an insertion sort of *n* items always requires exactly *n*-1 passes through the sorted portion of the list. What varies is the number of comparisons that must be performed per pass. The best case requires only one comparison per pass and occurs when attempting to sort a list that is already sorted. The worst case requires an average of *n*/2 comparisons per pass and occurs when the list is already sorted in *reverse* order. In the average case, only one half of the items in the sorted portion of the list will be examined during each pass before the insertion point is found – giving *n*/4 comparisons per pass.

Comparing sorts

When comparing the insertion sort to other sorts, generally the average case formula is used since this represents the expected performance of the algorithm. Occasionally, knowledge of the worst case behavior of the algorithm is also important. Understanding this behavior is useful when attempting to determine or limit the maximum amount of time a computing system will take to reach an answer, even in the worst case. Such behavior is important in real time applications such as airplane flight control systems.

The bubble sort and the selection sort always require exactly 1/2(n^2-n) comparisons to sort *n* items. In the worst case, the insertion sort also requires 1/2(n^2-n) comparisons. In the average (expected) case, however, the insertion sort requires 1/4(n^2-n) comparisons, and therefore requires about one half of the comparisons needed by the bubble and selection sorts. Figure 2 shows the runtime comparison of the three sorts, considering a machine capable of performing 1 million comparisons per second. The smaller number of comparisons needed by the insertion sort means that it is generally a faster algorithm than the bubble or selection sorts, assuming a comparison takes the same amount of time in both

algorithms (a reasonable assumption). The insertion sort will be expected to process a 10,000 item list in about 25 seconds (precisely 24.9975 seconds). The bubble and selection sorts are both expected to take about 50 seconds on the same problem (precisely 49.995 seconds) – or about twice as long.

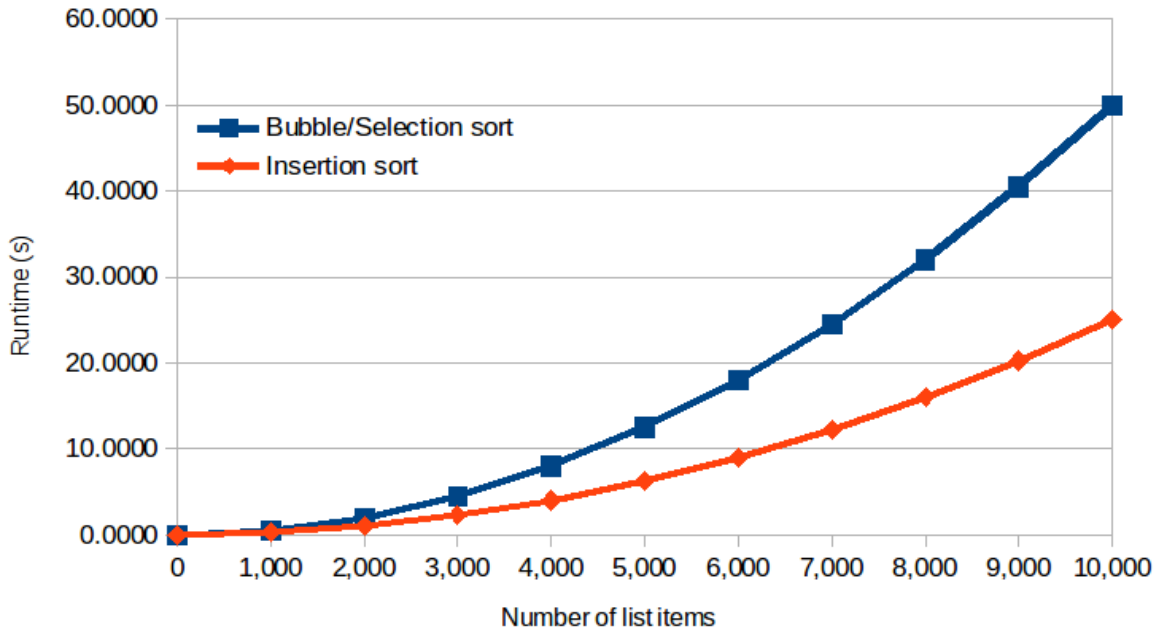


Figure 2: Runtime comparison of bubble, selection, and insertion sorts

The need for data structures

The algorithms we design to solve problems rarely do so without requiring some sort of input and producing some sort of output. In the process, our algorithms do something with the inputs (e.g., number crunching, processing, and so on). That is, algorithms typically manipulate the inputs in some way. Think about what that means. Where is the information? What does it *look* like? How is it accessed? How is it manipulated? Where does it go? We generally refer to the inputs being processed and the output(s) being generated as data.

Definition: ***Data*** *is a term given to pieces of information that can be represented, stored, or manipulated using a computer. Often, combining data provides meaning (i.e., information).*

Although data is useful and necessary, it is not yet meaningful. The idea is that our algorithms will process this data and produce some sort of output (or result). In the process, meaning is given to the data. We call this information.

Data structures

Data structures have to do with arranging or organizing data in some way. The memory capacity in today's computers is very large. Within the computer, data is stored in different memory locations. Often, the many pieces of data that our algorithms are dealing with are related in some way. Consequently, there is a need to have this data grouped in some way in memory. This grouping makes manipulation of that data much easier. Think about sorting a list of numbers. It would be much more difficult if the numbers were located randomly in memory. Somehow, we would need to know where each value is located, and that could technically be anywhere! Perhaps it would speed things up if each value was located in consecutive memory locations (i.e., next to each other in memory). We would then only need to know where the first value is located and the total number of values stored.

Definition: *A **data structure** is a way of organizing data in a computer so that it can be used efficiently.*

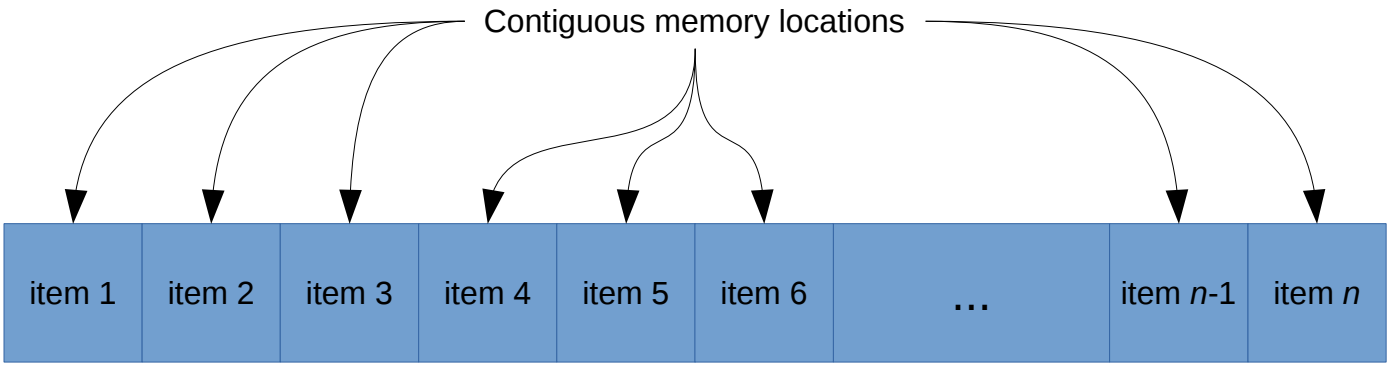
Data structures allow a programmer to arrange pieces of information (data) in a way that makes sense and allows the computer to manipulate the data easily for the programmer's task. Many times this data is made up of multiple instances of similar pieces of data, and other times it involves different kinds of data that are otherwise related. For example, a word (or a bunch of letters strung together) is made up of multiple pieces of similar data kept together in a specific order. In the case of a word, the letters of the alphabet are the pieces of data, and they have to be kept together in a specific order for it to make sense. Another example is a class roster which is made up of different entries in a specific order corresponding to each student in the class. Each entry is made up of dissimilar data such as the name and the grade on exam 1 of a student.

The array

One of the most commonly used data structures is called the **array**. We will see that many concepts in computer science, and particularly in data structures, are derived from real life examples – and arrays are not an exception. Arrays are comparable to a numbered list – such as a grocery list, a class roster, or a set of numbered drawers. They are used to store multiple instances of anything, as long as they are all of the same kind (i.e., all numbers, all letters, all images, all books, etc). Imagine these things being in

some sort of order (i.e., we have a first thing, a last thing, and some number of things in between). The members of (or entries in) the array are called elements.

Definition: *An **array** is a collection of similar pieces of data stored in contiguous memory locations. Contiguous memory locations means that the data is stored in memory locations that are next to each other.*



The order in which elements are stored in an array is important. This is because very often a programmer needs to access a specific element of an array, and in order to do that, its position relative to the first element of the array must be known. The position of an element is also referred to as its *address*, and the relative address (how far away from the first element it is) is called its *index*. We say that a *value* is stored at an *index* of the array.

The distinction between a **value** and its **index** is one that must be emphasized. In this context, a value refers to a piece of data stored in an array, and its index is the position in the array where that value is stored. The index represents *where* an element is, and the value represents *what* the element is. While the two are related, each of them will be of different importance to us depending on the scenario we are trying to solve. For example, if you misplaced your favorite jacket at home, its location would be more important than its value (i.e., its index would be more important than the fact that it is a jacket). In contrast, when you get feedback on a test you did in class, the value of your score is more important. While dealing with arrays, it is important to understand the distinction between index and value.

The Python sequence

The most basic data structure in Python is the sequence. A **sequence** is composed of (related) elements. Each element in a sequence is assigned an index (or position). A sequence with n elements has indexes 0 to n–1. Pay special attention to this! The first value in a Python sequence is located at index 0, and the n^{th} (i.e., last) value is located at index n–1 . Python has many built-in types of sequences; however, the most popular is called the list. It effectively functions as an array!

The **list** in Python is quite versatile and is declared using square brackets; for example:

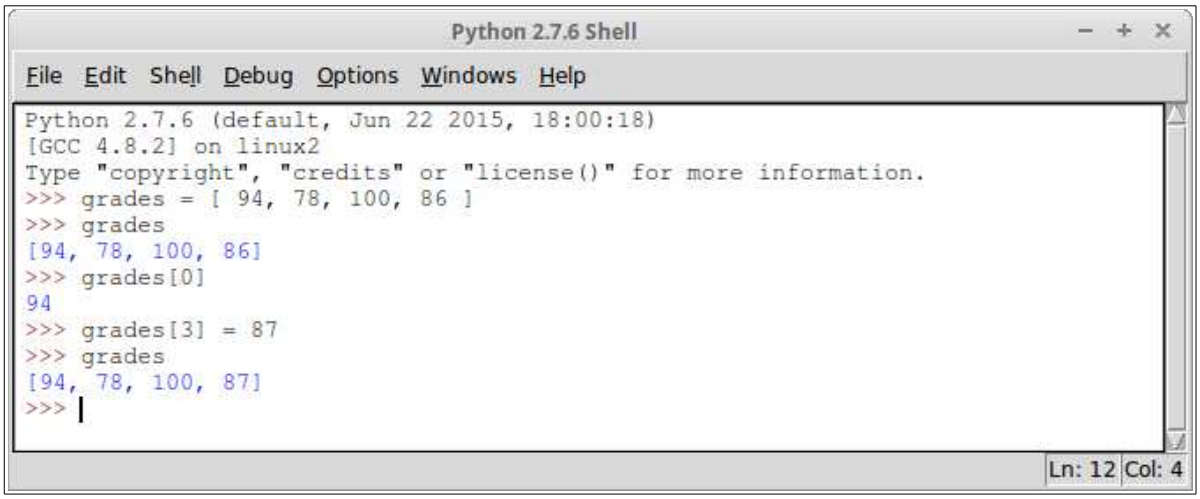
```
grades = [ 94, 78, 100, 86 ]
```

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

```
grades[0]
```

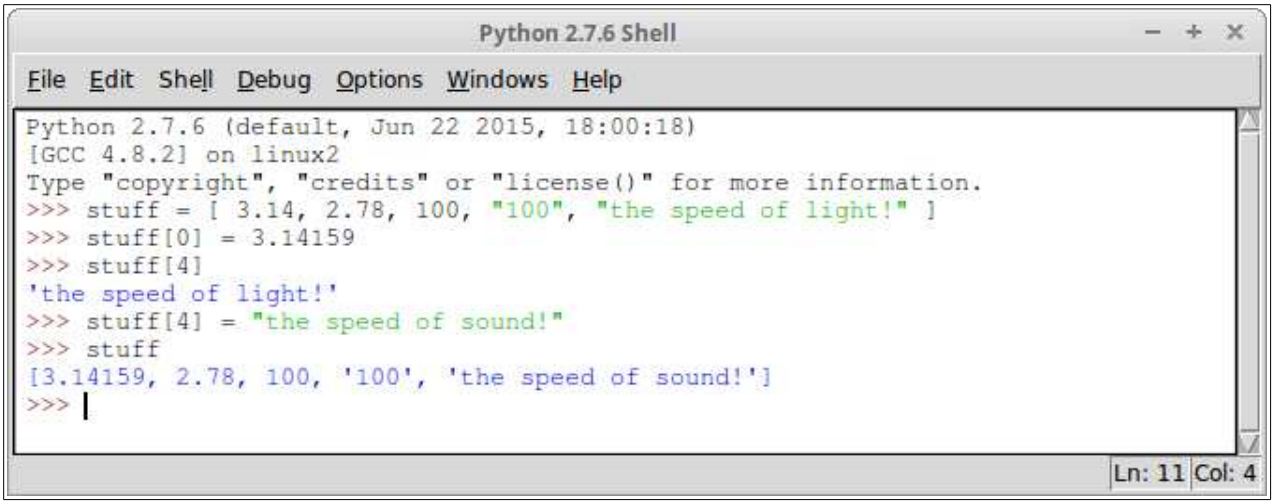
```
grades[3] = 87
```

Here's an example of this:



Note that, in Python, the values within a list do not need to be of the same data type! This is a bit different than lists in most other general purpose programming languages (usually, those languages call their lists **arrays**), in which all elements must be of the same type. Here's an example of a *heterogeneous* (meaning diverse) list in Python:

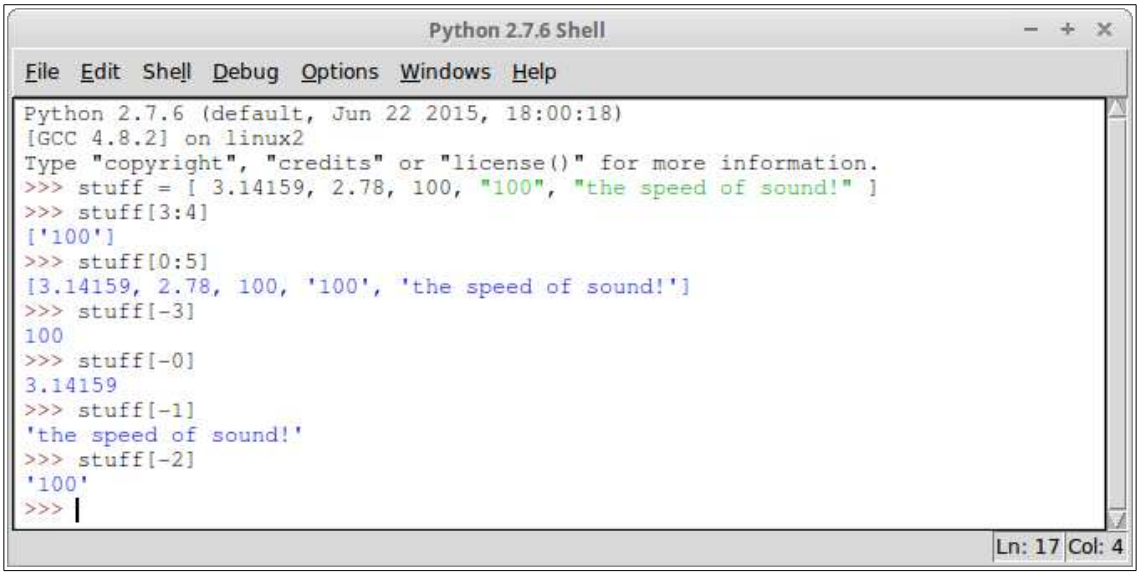
```
stuff = [ 3.14, 2.78, 100, "100", "the speed of light!" ]
```



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

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

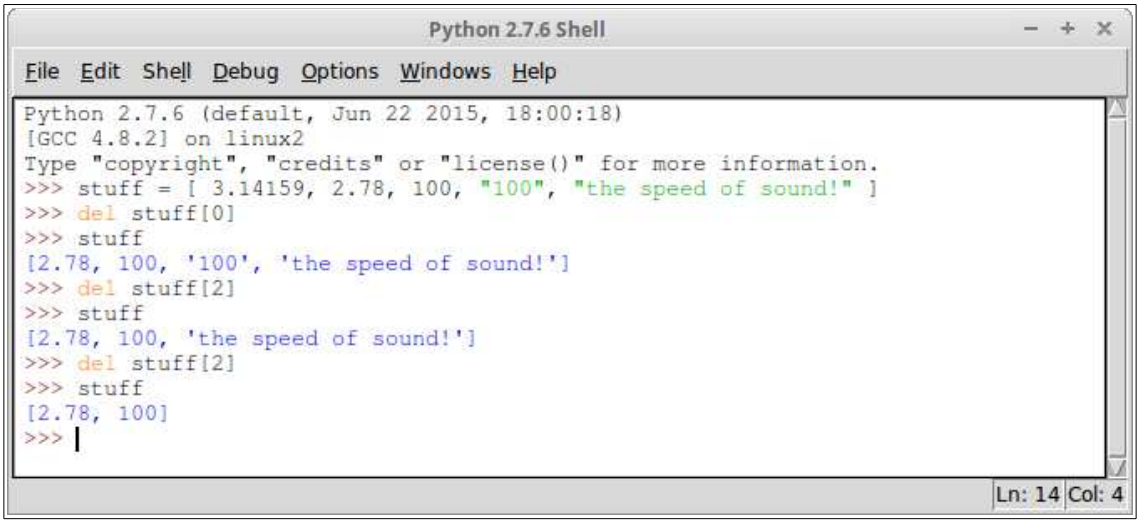
Here are some examples:



Note the difference between the element `100` (a number) at index 2 and the element `'100'` (a string) at index 3.

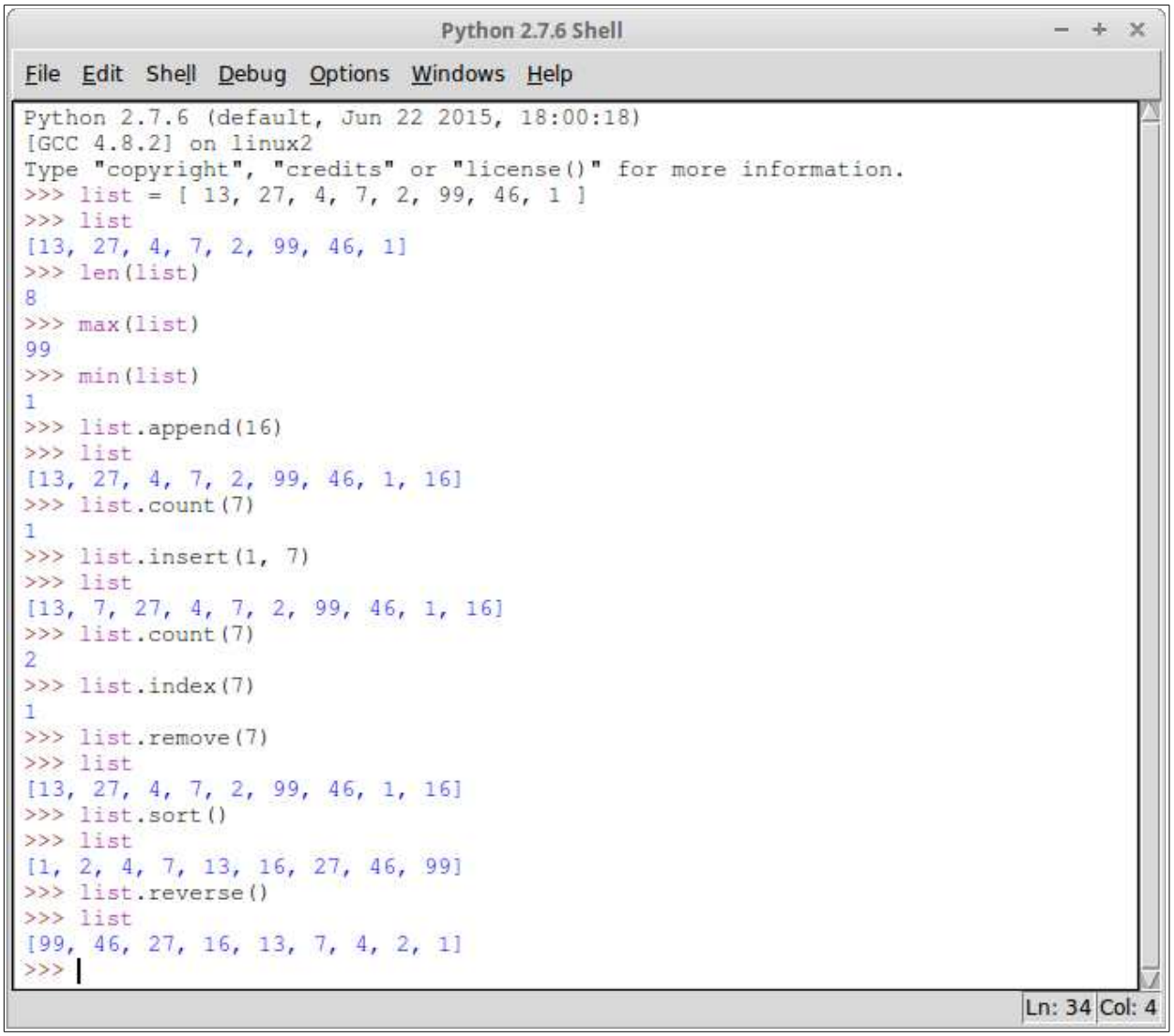
List elements can be deleted with the **`del`** keyword as follows:

```
del stuff[2]
```



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

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



Activity 1: Creating and populating an array (actually, a Python list)

In this activity, you will write a Python program to create a list of 20 random integers.

Creating the list
Create an empty list named *numbers* using a simple assignment statement:

```
numbers = []
```

Currently, the list is empty (i.e., its length is 0). In fact, this can be confirmed using one of the list methods shown above:

```
print len(numbers)
```

Populating the list with random integers
Let's populate the list with 20 random integers. While it is possible (albeit tedious) to populate the list by adding each value individually, instead we are going to create a short program to make the population process automated.

Because the population process calls for a specific task (i.e., the addition of an item to the list) to be repeated over and over again, let's use the **while** loop with a counter (initialized to 0). Let's now declare the counter. Note that statements already shown above are included, and new statements are highlighted:

```
numbers = []
counter = 0
```

Because our task is going to be done 20 times (in order to fill the list with 20 values), we can structure the while loop as follows:

```
numbers = []
counter = 0
while (counter < 20):
    # add a random number to the list
    counter += 1
```

Note that the comment (beginning with #) will be replaced later. It is just informative at this point. Notice how the counter is incremented at the end of the while loop. This is necessary because it allows it to progress from 0 to 20, guaranteeing that the condition in the while loop will eventually be false (i.e., *counter* will be greater than or equal to 20). This stops the repetition, allowing control to continue past the while loop.

Now, what number will we be actually adding to the list? Well, we want to add 20 *randomly* selected numbers to the list. To generate random numbers, we can make use of a Python library called **random**; specifically, a function in the library called **randint**. Libraries and how to import them in Python programs will be discussed in more detail in a later lesson. For now, we'll simply show how to import the **randint** function from the **random** library:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
```

```
        # add a random number to the list
        counter += 1
```

Next, let's generate a random integer from 1 to 99:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    num = randint(1, 99)
    counter += 1
```

Note that the **randint** function takes two parameters: a lower bound and an upper bound. The function returns a random integer from the lower bound to the upper bound, inclusive. So far, we've just generated 20 random numbers from 1 to 99. We're not actually adding them to the list! Let's do this now using the **append** list function shown above:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    num = randint(1, 99)
    numbers.append(num)
    counter += 1
```

You have now created a list (which we called *numbers*) and filled it with 20 random integers. If any of the following activities require a list filled with random numbers, you can easily refer to the above steps and create one. If you require a larger or smaller array, its just a matter of changing the value in the **while** loop. If you require values in a different range (perhaps numbers between 100 and 1000), its just a matter of changing the values in the **randint** function.

The generated list can be displayed as follows:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    num = randint(1, 99)
    numbers.append(num)
    counter += 1

print numbers
```

The output generated will look something like this (of course, your list will look different):

```
[82, 17, 85, 8, 2, 7, 33, 13, 24, 89, 49, 37, 61, 72, 83, 39, 23, 58, 45, 31]
```

You may wonder why we store a randomly generated integer in the variable *num* in the program above. It's not used for anything other than being added to the list. In fact, the variable *num* is actually not necessary. The algorithm above can be modified without changing its behavior as follows:

```
from random import randint

numbers = []
counter = 0
while (counter < 20):
    numbers.append(randint(1, 99))
    counter += 1

print numbers
```

The randomly generated integer is generated and immediately added to the list!

As a last observation, it is often good practice to use constants to specify target or desired values. For example, using a constant to specify the list's desired size can be useful. Any point in the program that needs the list's target size can refer to the constant. If it needs to be changed at a later time, the constant value simply needs to be changed, and all references to it through the constant won't have to also be updated. Here's a modification of the above program that implements this:

```
from random import randint

SIZE = 20
numbers = []
counter = 0
while (counter < SIZE):
    numbers.append(randint(1, 99))
    counter += 1

print numbers
```

Finally, recall that one of the list methods shown above returns the list's size. We can use it instead of a counter to repeatedly insert integers into the list until it has reached the desired size:

```
from random import randint

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

print numbers
```

Note that all statements referring to the variable counter have been removed. As integers are inserted into the list, its length increases. Therefore, `len(numbers)`, initially 0, increases by one each time an integer is inserted into the list. Eventually, `len(numbers)` will be 20, and control will continue past the while loop.

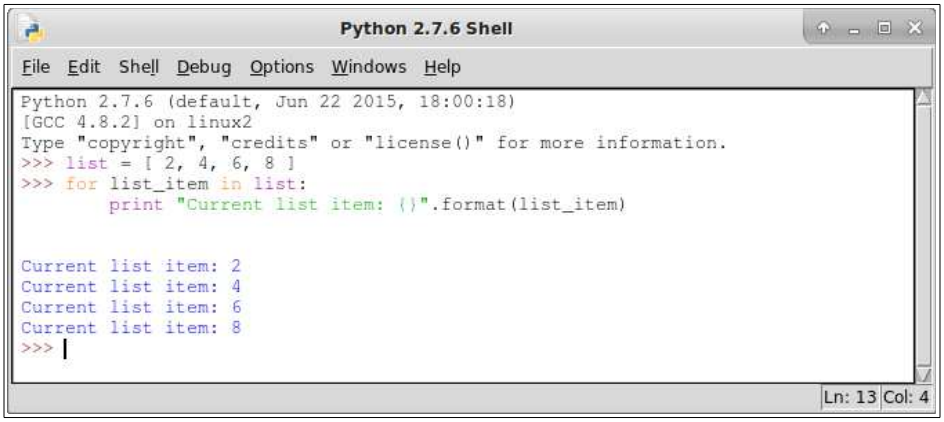
The Python for loop

Python provides one more repetition construct called the **for** loop. Typically, while loops are considered to be sentinel-driven. That is, they require a condition to either be true or false that indicates execution of the statements in the loop. Recall that Scratch also had a **repeat-n** block that repeated a task a set (or fixed) number of times. This kind of loop can be implemented in Python using the for loop construct.

The structure of a for loop in Python is:

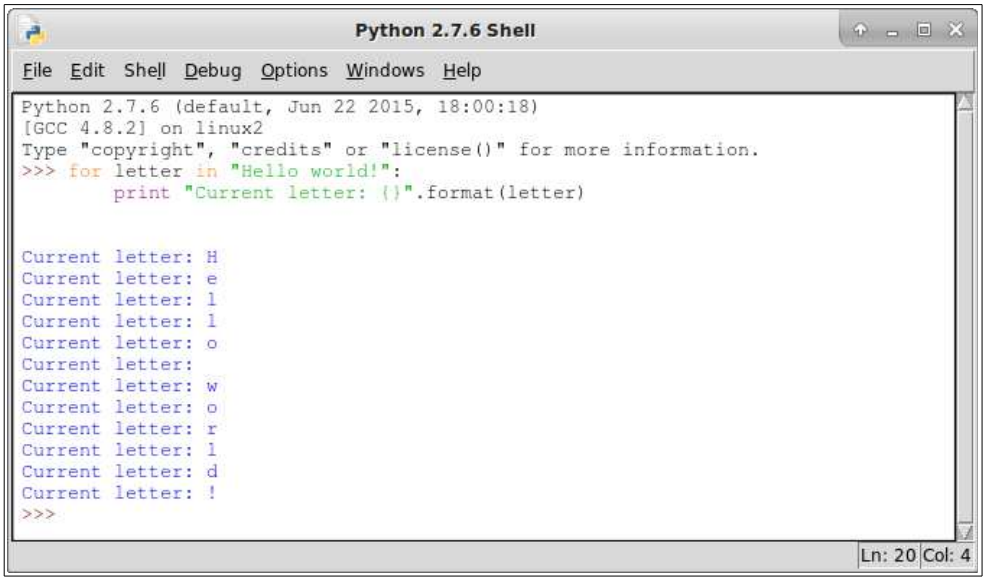
```
for iterating_variable in sequence:
    loop_body
```

Here's an example that uses the for loop on a list:

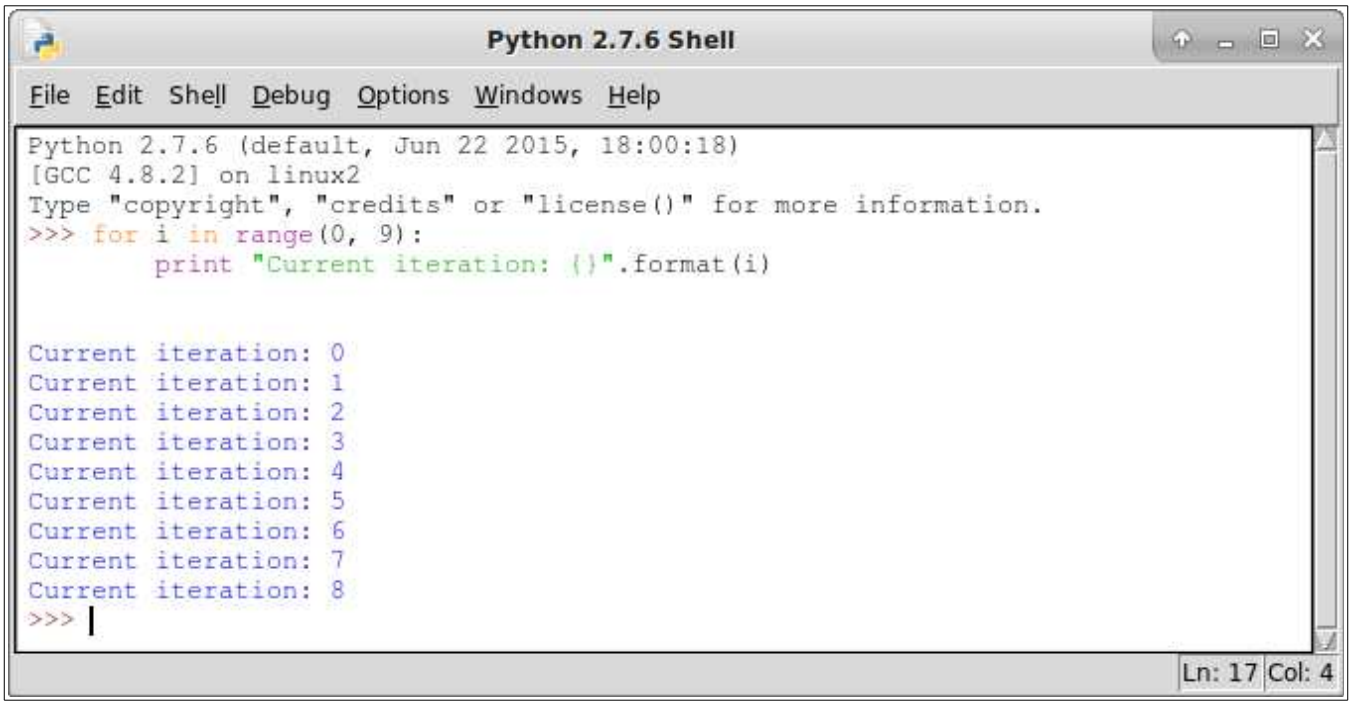


The variable `list_item` is used as an iterator. That is, it takes on the value of each item in the list at each iteration of the for loop. The first time, `list_item` takes on the first item in the list, 2. The second time, it takes on the second item in the list, 4. Eventually, it takes on the last time in the list, 8. In total, the body of the for loop executes once for each item in the list (or four times).

The for loop is actually quite powerful and flexible. It can, for example, iterate through the letters of a string:



Typically, for loops iterate through a counter. The counter can then be used to refer to a variety of things, including the index of the elements of a list. To structure a for loop such that it iterates, say from 0 through 8, we would use the built-in Python function, `range()`:



The `range()` function typically takes two parameters: a start value and a stop value. The start value is included in the range; however, the stop value is not. For example, to iterate from -10 to 10:

```
for i in range(-10, 11)
```

Note that an optional third parameter can be specified to indicate a step value. For example, suppose that you wish to iterate from 0 to 100 in increments of 10 (i.e., 0, 10, 20, …, 90, 100):

```
for i in range(0, 101, 10)
```

More on the for loop will be discussed later.

Now that we have a list populated with random values, we can implement the sequential search to find the largest value in the list!

Activity 2: Sequential search of the largest value in an array

This activity involves implementing a sequential search on the list that was created in the first activity. We will use the sequential search to find the **largest** value in the list. We need to use our knowledge of the sequential search algorithm to implement it using the tools available to us in Python.

Just to jog your memory, the sequential search algorithm begins by assuming that the largest value is the value stored in the first position of the list. We then check through every position, one by one, to see if there is a value greater than what is currently stored as largest. If a larger value is found, then the largest is replaced with the value in the current position. Otherwise, we just move on to the next position. This process is repeated until the end of the list is reached.

The first thing to do is to create a variable called **largest** and assign it the value in the first position of the list:

```
largest = numbers[0]
```

Now that we have initialized the variable **largest**, we need to compare it with each value in the list. This sounds like we'll need another loop construct (to do the comparison 20 times: one comparison for each of the values in the list). Technically, we really only need to do the comparison 19 times: one comparison for each of the remaining values in the list). Let's use a **for** loop for this.

Recall that the sequential search algorithm dictates that if the value we are comparing with in the list is greater than **largest**, then we change the value of **largest** to store the current value in the list. We then keep on comparing it with the remaining values in the list.

To compare the variable **largest** with individual elements in the list, we will need to know the indexes of each of the values in the list. That is, we will need to start with the second index (why not the first?), compare it with largest, go to the third, and so on. The strategy will be to use a counter to iterate through the range of indexes in the list and update the variable largest as necessary. Recall that the first index in the list is 0. Since we assume that the largest value is currently there, we can begin searching at index 1:

```
largest = numbers[0]

for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]
```

The function **range(1, 20)** generates the sequence 1 through 19 that the variable **index** will iterate through (i.e., take on the value 1, followed by 2, and so on). We can therefore use the variable **index** to refer to specific locations in the list and access each element in the list. The benefit of using a variable instead of an explicit number is that a variable can change at any point in our algorithm.

The last step is to display the largest value:

```
largest = numbers[0]

for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]

print largest
```

Perhaps a more meaningful output statement is better:

```
largest = numbers[0]

for index in range(1, 20):
    if (numbers[index] > largest):
        largest = numbers[index]

print "The largest value is: {}".format(largest)
```

The entire program, including generating the list, is shown here for completeness:

```
from random import randint

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

print numbers

largest = numbers[0]

for index in range(1, 20):
     if (numbers[index] > largest):
          largest = numbers[index]

print "The largest value is: {}".format(largest)
```

Activity 3: Selection sort of an array

For this activity, you will need a list of 20 random integers from 1 to 100. Refer to the Python code in the activities above if necessary.

Now, let's implement the selection sort. First, let's look back at the pseudocode for the algorithm:

```
 1: n ← length of the list
 2: for i ← 0..n-1
 3:     minPosition ← i
 4:     for j ← i+1..n
 5:         if item at j < item at minPosition
 6:         then
 7:             minPosition ← j
 8:         end
 9:     next
10:     swap items at i and minPosition
11: next
```

The selection sort works by building a sorted list from left-to-right. Initially, the smallest value is located in the list and swapped with the first item in the list. The sort repeats this process with the next unsorted element (i.e., the first item in the unsorted portion of the list). Each iteration, a *minPosition* is updated that reflects the position (or index) of the smallest value in the list so far. Once the entire list has been searched through, a swap is made (swapping this minimum value with the first value in the unsorted portion of the list).

Here is the selection sort in Python (using the list **numbers** generated above):

```
n = len(numbers)

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

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

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

Notice how the values at indexes *i* and *minPosition* of the list are swapped. A **temporary** variable is used to store on of the values, thereby allowing that value to be overwritten with the one to be swapped with. In other words, to swap two values, *x* and *y*, we temporarily store *x*, overwrite *x* with *y*, and overwrite *y* with the temporarily stored *x* (i.e., *temp* = *x*, *x* = *y*, *y* = *temp*).

Let's break the algorithm down, step-by-step. The first statement is pretty evident and is easily translated to Python: `n ← length of the list` becomes `n = len(numbers)`.

The selection sort makes use of two loops, one inside the other. The outer loop controls the number of passes made through the list, each time placing the next smallest value in the list. The inner loop finds the next smallest value by comparing each value in the list to the current minimum.

```
 2: for i ← 0..n-1
 3:     minPosition ← i
 4:     for j ← i+1..n
 5:         if item at j < item at minPosition
 6:         then
 7:             minPosition ← j
 8:         end
 9:     next
10:     swap items at i and minPosition
11: next
```

It's very clear where both for loops are in the Python code:

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

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

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

In fact, the pseudocode and Python code are nearly identical! The initialization of *minPosition* to *i* is almost the same: `minPosition ← i` becomes `minPosition = i`.

The if statement in the inner loop looks slightly different; however, that's just because of how list items are accessed in Python: using an index enclosed in square brackets. The relevant pseudocode is:

```
 5:  if item at j < item at minPosition
 6:  then
 7:       minPosition ← j
 8:  end
```

And in Python:

```
if (numbers[j] < numbers[minPosition]):
     minPosition = j
```

As mentioned earlier, swapping is typically done by declaring a temporary variable, assigning it one of the two values to be swapped, and then performing successive assignments. Therefore, although the following pseudocode is nice and meaningful, it is not directly translatable to Python:

```
swap items at i and minPosition
```

Instead, we use successive assignment statements:

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

The selection sort algorithm in Python stores the value at index *i* of the list in a temporary variable (`temp`). It then replaces the item at index *i* of the list with the item at *minPosition*. Finally, it replaces the item at index *minPosition* of the list with `temp` (the item that used to be at index *i* of the list).

Now that we have implemented the selection sort, we can sort any array of values! And now that we have a sorted array, we can implement a more efficient search than the sequential search. Recall that there exists a more efficient search that only works on sorted data: the binary search. Let's try to implement it now in the next activity.

Activity 4: Binary search for a specific value in an array

For this activity, we assume that you have declared and randomly populated a list of values (called **numbers**), and that you have also sorted this list using the selection sort implemented in the previous activity.

Let's begin by recalling the pseudocode for the binary search as shown in a previous lesson:

```
 1: repeat
 2:     n ← number of items in the current portion of the list
 3:     mid ← floor(n / 2) + 1
 4:     guess mid
 5:     if response is HIGHER
 6:     then
 7:         discard the left half of the list
 8:     else if response is LOWER
 9:     then
10:         discard the right half of the list
11:     end
12: until guess is correct
```

Note that the repeat-until loop terminates when the guess is correct. This is because we tailored the binary search to the number guessing game (which means that, eventually, the number will be found). We need to generalize the algorithm so that it can work on an arbitrary list of values, whether the specified value is found in it or not. Try to rewrite the algorithm so that it works for an arbitrary list.

This new version works by finding the middle value in the list. It then compares the value to search for with this middle value. If they match, then the search is finished! Otherwise, the left or right half of the list is discarded, depending on the result of the comparison. This continues, either until the value is found, or until the entire list has been searched (and the value was not found).

Generally, it's not a good idea to actually remove values from a list when performing a search. We can tweak the algorithm a bit to maintain the list's integrity by keeping track of the "beginning" and "end" of the **valid** portion of the list. That is, if we wish to "discard" the left half of the list, then the valid portion of the list is the **right half**: the beginning of the valid portion of the list is at the index directly to the right of the middle value, and the end of the valid portion of the list doesn't change. And if we wish to "discard" the right half of the list, then the valid portion of the list is the **left half**: the beginning of the valid portion of the list doesn't change, and the end of the valid portion of the list is at the index directly to the left of the middle value. Here's a modified algorithm that implements this tweak:

```
num ← number to search for
found ← false
first ← 0
last ← number of items in the list - 1
while first <= last and found != true
    mid ← floor((first + last) / 2)
    if num = item at mid of the list
    then
        found ← true
    else if num > item at mid of the list
    then
```

```
            first ← mid + 1
        else
            last ← mid - 1
        end
    display found
```

If the desired value is found in the list, the search terminates. This occurs because the variable *found* is set to true when the value is found in the list, and the while loop's condition requires `found != false` to continue the repetition. The while loop also terminates if the variables *first* and *last* shift sides (i.e., *first* represents an index in the list that is greater then or equal to *last*). This indicates the the desired value was not found in the list.

Note how the variable *mid* is calculated: it is the average of the first and last indexes! The **floor** math function guarantees that the middle value is to the left of the center of the valid portion of the list, if the valid portion of the list has an **even** number of items.

Here's a Python implementation of the binary search:

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

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

print found
```

First, the desired search value is obtained and stored into the variable *num*:

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

Next, *found* is initialized to false, *first* to 0 (the first valid index in the list), and *last* to `len(numbers) - 1` (the last valid index in the list):

```
found = False
first = 0
last = len(numbers) - 1
```

The while loop is structured so that it iterates so long as `first <= last` (i.e., the beginning and end of the valid portion of the list haven't logically swapped) and `found != True` (i.e., the desired value hasn't already been found):

```
while (first <= last and found != True):
```

Next, the middle index of the valid position of the list is calculated as the average of *first* and *last*. Note the use of the `//` operator (which performs a floor division):

```
mid = (first + last) // 2
```

If the desired value is found at the index stored in *mid*, the *found* is set to true (which will terminate the while loop). If not, then either *first* or *last* is appropriately updated: if the desired value is greater than *mid*, then *first* is updated to *mid* + 1; if the desired value is less than *mid*, then *last* is updated to *mid* – 1.

```
if (num == numbers[mid]):
     found = True
elif (num > numbers[mid]):
     first = mid + 1
else:
     last = mid - 1
```

To assure yourself that the loop will terminate if the desired value is not found by logically swapping the positions of *first* and *last*, let's try a trivial example with a list of one item: `[ 5 ]`.

Running the binary search on this list initially sets *found* to false, *first* to 0, and *last* to 0 (`len(numbers) - 1` = 1 – 1 = 0). At this point, *first* is indeed less than or equal to *last* (actually *first* == *last*). Moreover, *found* != true. Therefore, *mid* is calculated to be (0 + 0) // 2 = 0 // 2 = 0.

Suppose that we wish to search for the value 6 (which is not in the list). Therefore, *num* != *numbers*[*mid*]; however, *num* > *numbers*[*mid*] (i.e., 6 > 5). This results in *first* being updated to 1: *mid* + 1 = 0 + 1 = 1. When the condition in the while loop is reevaluated, *first* is 1 and *last* is 0! The values have logically swapped (i.e., how can *first* be greater than *last*?). Therefore, the while loop terminates, and the variable *found* is never being changed from its initialized value of false.

How could the algorithm be modified so that the output is more meaningful? That is, if the desired value is found, additionally display its index in the list. A possible solution is:

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

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

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

The reason that *mid* can be used to provide the index of the desired (and found) value in the list, is that it is not recalculated when *found* is set to true. That is, its last calculated value is preserved (which is the last calculated middle of the valid portion of the list – where the desired value was found).

RPi Activities

Raspberry Pi Activity: Sampling Some Raspberry Pi

In this activity, you will learn about the platform used in the Living *with* Cyber curriculum. You will need the following items:

- Louisiana Tech Living *with* Cyber kit

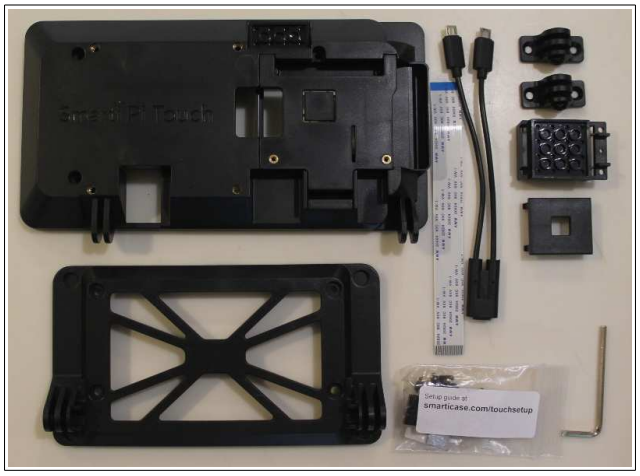
Opening the kit

The Louisiana Tech Living *with* Cyber kit includes the Raspberry Pi (a small single-board computer), an LCD touchscreen, keyboard and mouse, and a variety of other components that will be used in activities throughout the curriculum. It comes in a carrying bag. Here are the kit parts:

The kit includes a USB keyboard and mouse to provide input to the Raspberry Pi, and a touchscreen display and speakers to provide output from the Raspberry Pi.

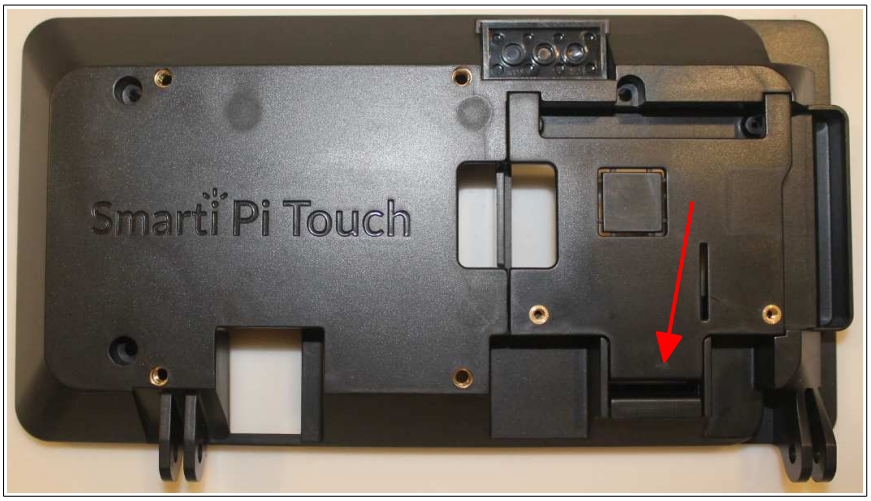
Assembly

To begin, unbox the Smart Pi Touch. We'll first build the stand that will hold the LCS touchscreen and Raspberry Pi:

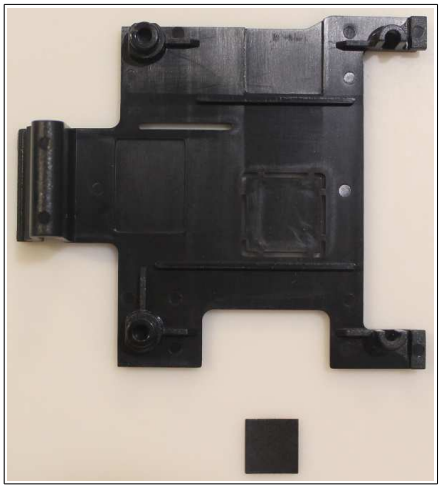


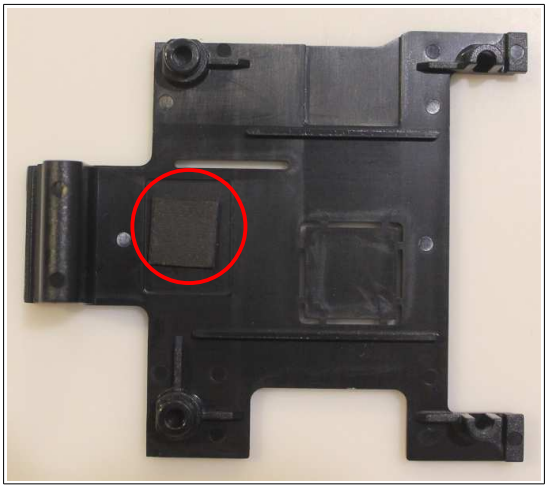
Start by sticking the four clear, round pads on the bottom of the stand:

Next, remove the door on the part of the stand that will hold the LCD touchscreen and Raspberry Pi:

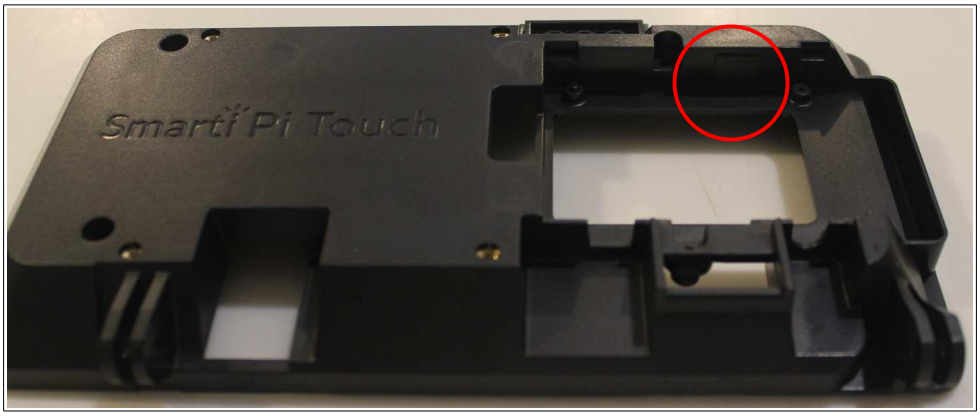


On the rear of the door, in the center of the marked rectangle, stick a black felt pad:

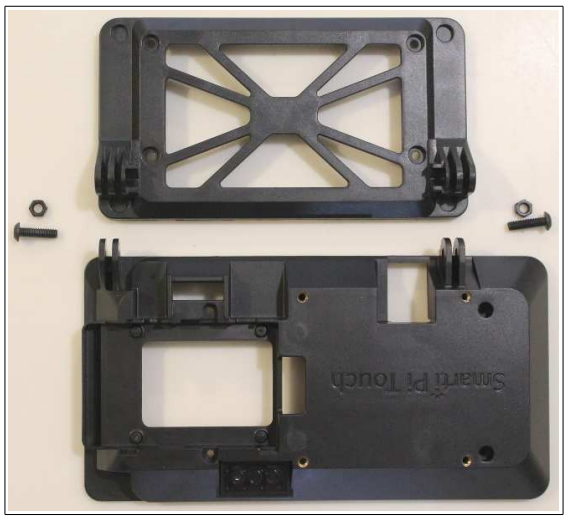




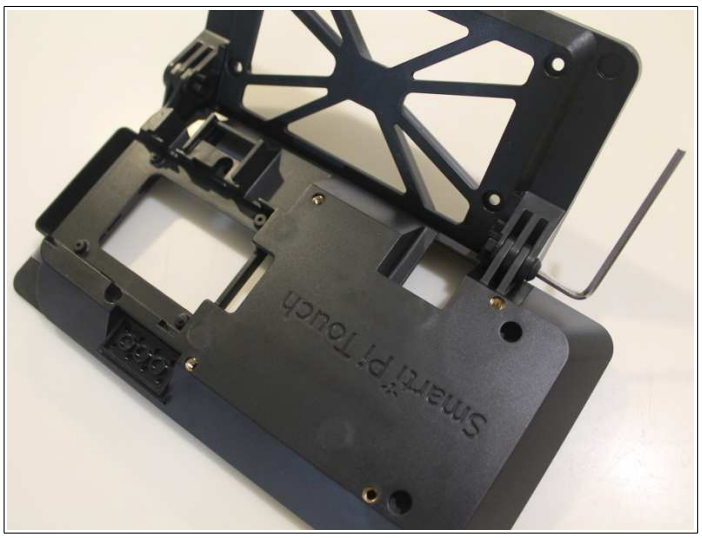
Do the same near the rear edge of the stand:



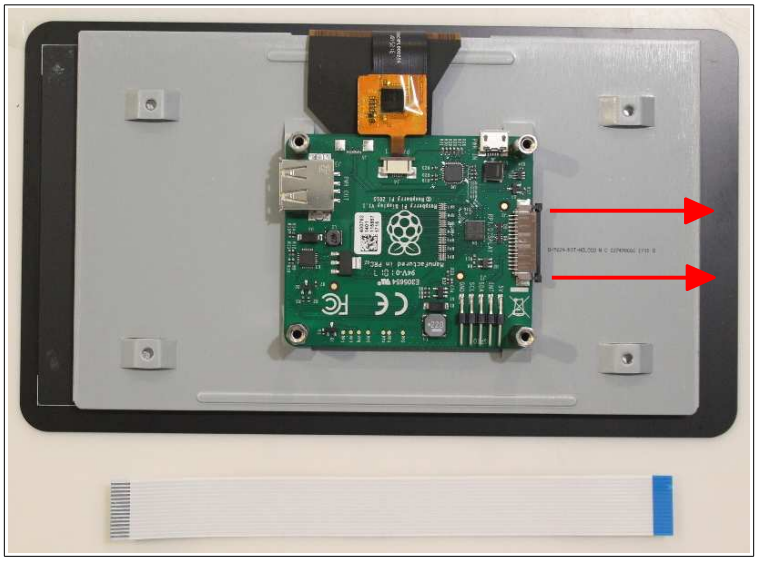
Next, connect the two parts of the stand together and secure with the two large black screws and nuts:



Tighten the nuts with the hex key (aka Allen key) provided with the stand. **Do not overtighten!**

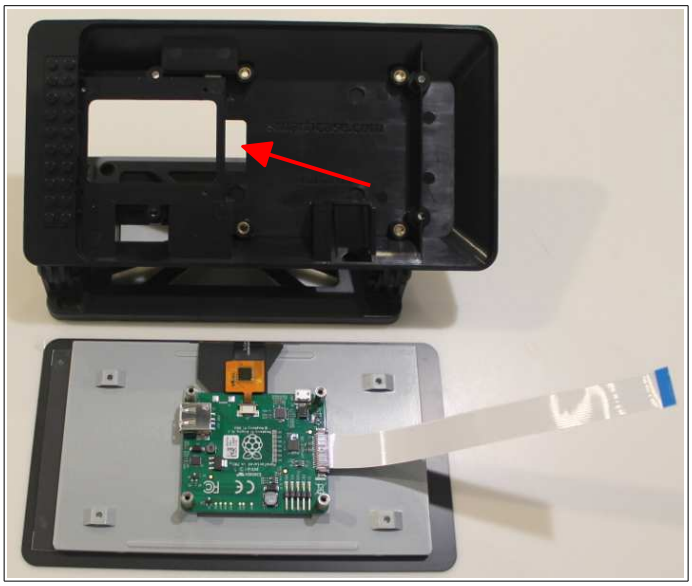


It's now time to install the LCD touchscreen to the stand. Remove the LCD touchscreen from its box. You'll also want the long ribbon cable that came with the stand handy. Pull out the gray tab as shown here:



Insert the ribbon cable, metal contacts facing up into the slot. Then, secure the gray tab to lock the ribbon cable in place:

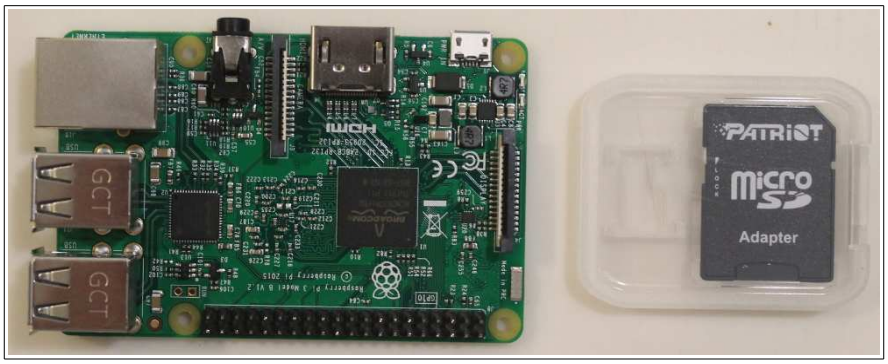
Next, place the LCD touchscreen into the stand (pay attention to the orientation), making sure to insert the free end of the ribbon cable into the slot in the stand:



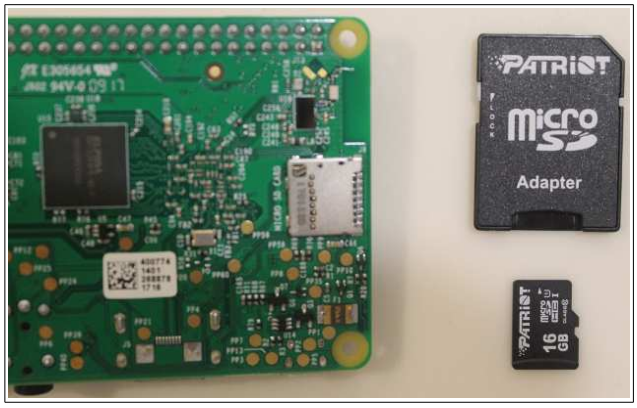
for the next step, you will need the Phillips screwdriver. The kits actually contains a single screwdriver with both a Phillips head and a flathead. The metal shaft is removable:

You can see below how the ribbon cable sticks out of the slot in the stand. Secure the LCD touchscreen with four small black screws as show below. You will need to hold the LCD touchscreen in place from beneath with your free hand:

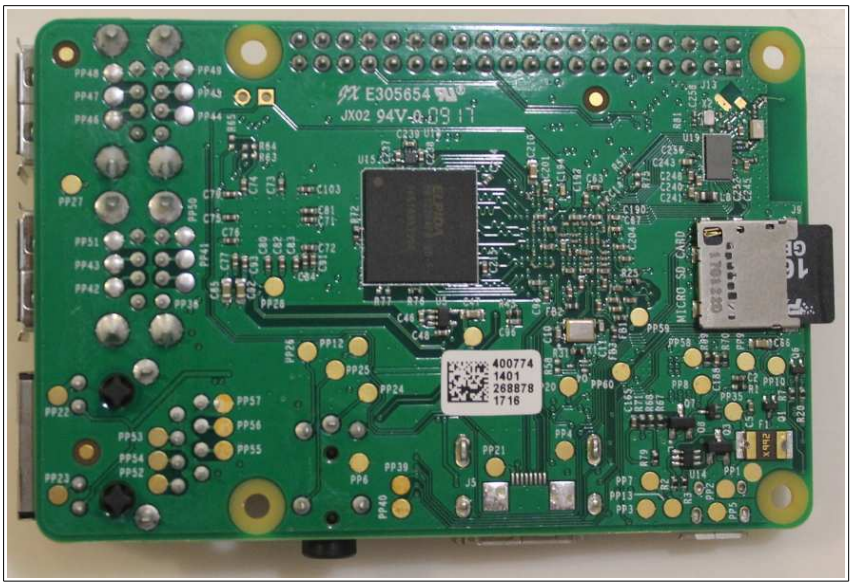
Remove the Raspberry Pi from its box. You will also need the microSD card (note that it comes inside a larger SD card adapter):



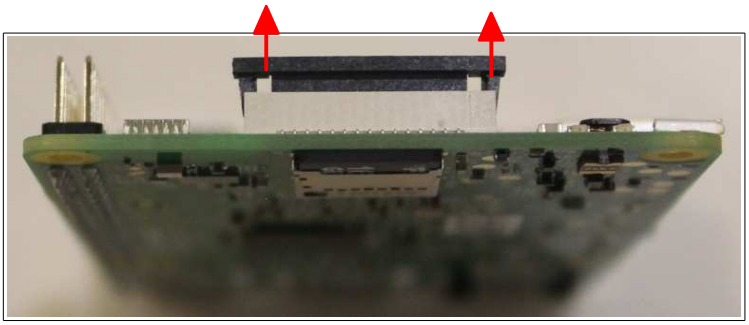
Remove the microSD card from the SD card adapter and turn the Raspberry Pi on its back:



Carefully insert the microSD card as shown below into its slot on the Raspberry Pi. **The microSD card is quite delicate and can be bent (and therefore broken) easily. Be careful!**



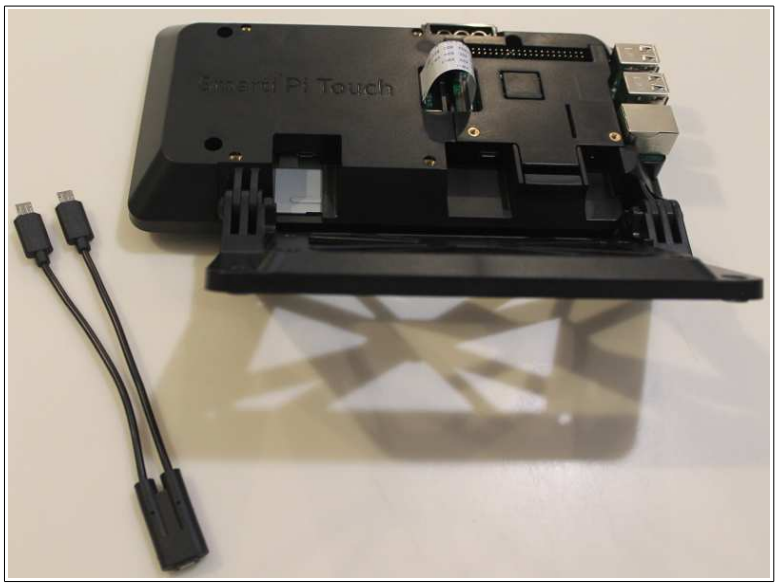
Slide the black tab on the front of the Raspberry Pi out so that the ribbon cable can be inserted later:



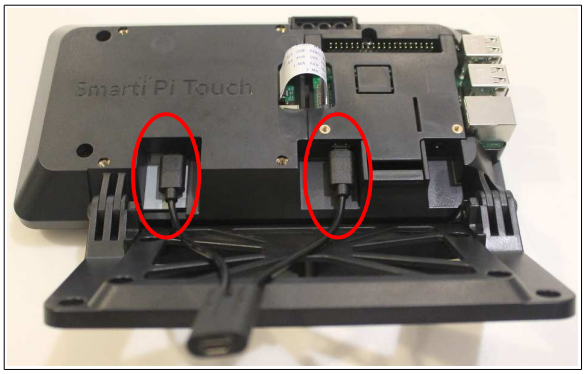
Insert the Raspberry Pi into the stand as show below, taking care to properly insert the ribbon cable in its slot on the Raspberry Pi. Make sure the slide the black tab back in to secure the ribbon cable to the Raspberry Pi:

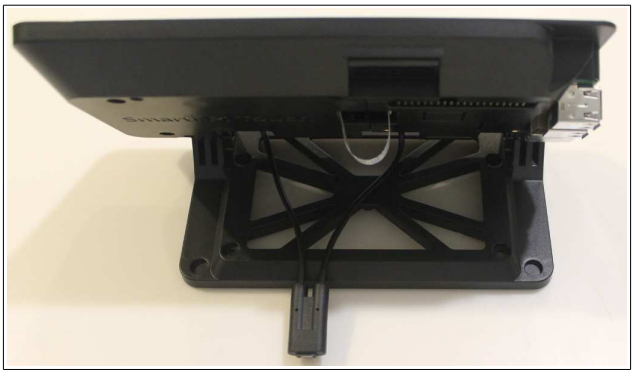
Finally, replace the door on the back of the stand, being careful not to damage the Raspberry Pi:

To wrap up this part, you will need the USB y-adapter. It will provide power to both the Raspberry Pi and the LCD touchscreen:



Connect it as shown below, then straighten the stand:



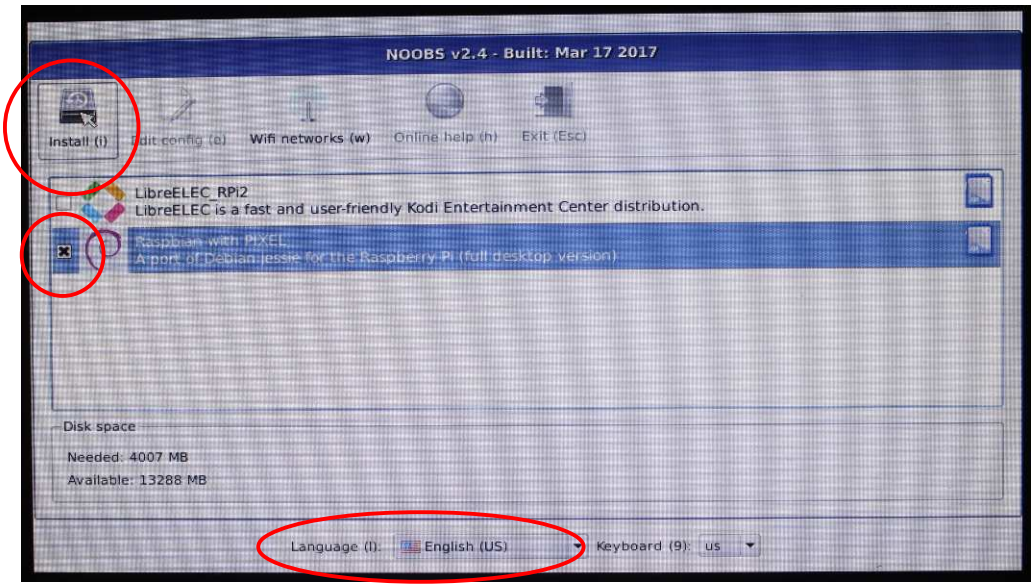


Remove the power adapter from its box and connect it to the USB y-adapter:

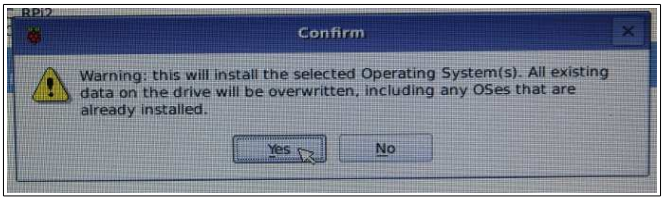
Finally, Remove the keyboard and mouse from their box and connect them to the Raspberry Pi via two of the USB ports:

Installation

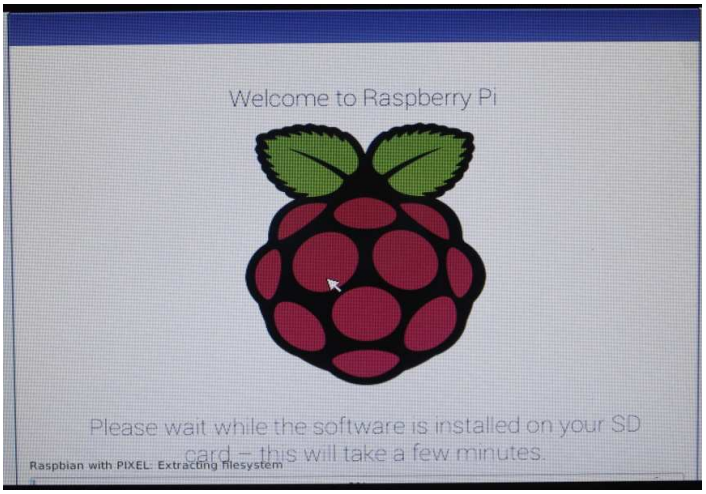
This process installs the operating system on the Raspberry Pi. Plug the power adapter into an outlet to turn the Raspberry Pi and touchscreen display on. When NOOBS (New Out Of the Box Software) loads, select the operating system listed: Raspbian with PIXEL and change the language to English (US) at the bottom. To apply the changes, click on the **install icon** near the top-left of the window:



Confirm the installation by clicking **Yes**:



At this point, the installation will begin (which may take some time to complete):



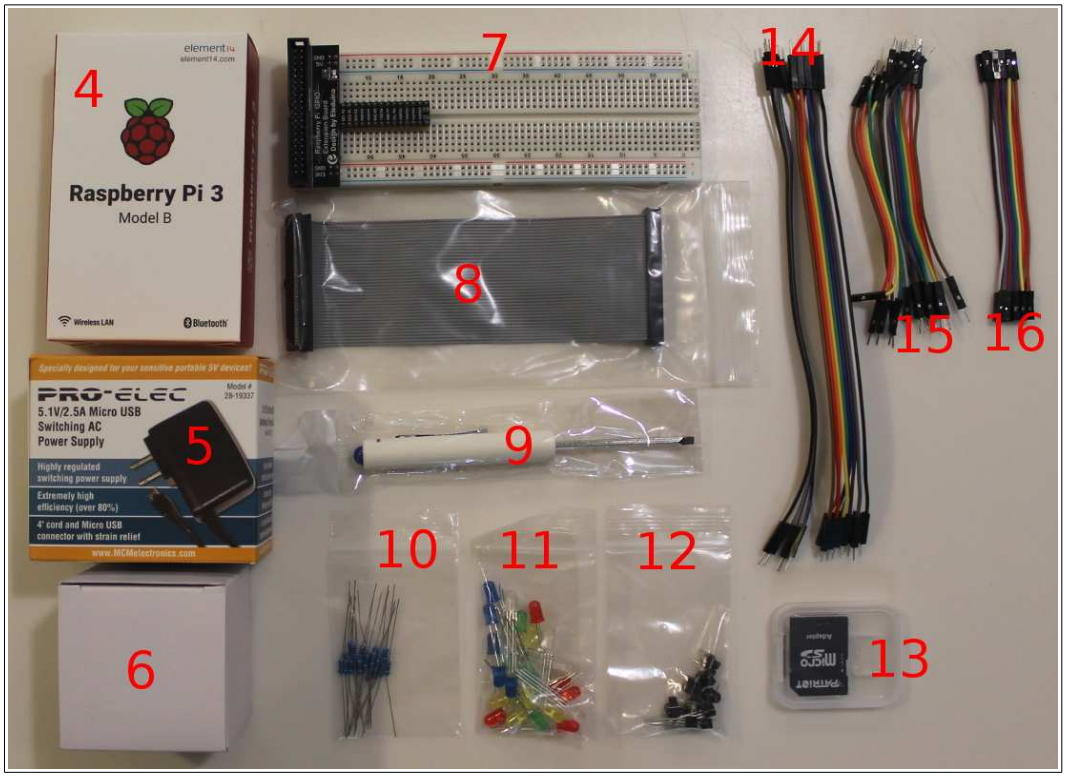
Once finished, you will see a confirmation message. Click **OK**:

What's in the kit?
While we're waiting for the installation process to complete, let's learn about the various components that make up the computing platform that is used in the Living *with* Cyber curriculum. Here are some of the bigger components:

Let's take a look at each numbered component in the figure above:

1. One **wired keyboard and mouse**. This is often preferred to the touchscreen capabilities of the display, especially when programming.
2. One **Smart Pi Touch stand** to secure the LCD touchscreen display and the RPi in a neat all-in-one solution.
3. One **7" LCD touchscreen display** that serves as the monitor.

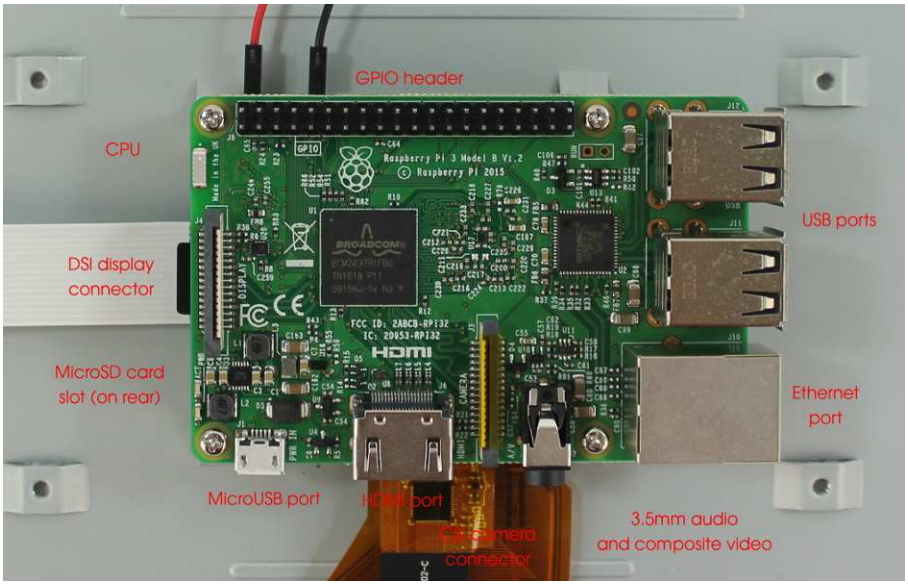
An now some of the smaller components:



Let's take a look at each numbered component in the figure above:

4. One **Raspberry Pi 3** (Model B) computer.
5. One 5V **power adapter** that provides power to the RPi.
6. One **USB-powered speaker** to hear audio coming from the Raspberry Pi (RPi).
7. One **breadboard** that provides space for prototyping circuits with one **GPIO-breadboard interface** that allows the GPIO pins on the RPi to be extended to a breadboard (more about this later).
8. One **GPIO ribbon cable** that connects the GPIO pins on the RPi to the GPIO-breadboard interface.
9. One Phillips and flathead **screwdriver**.
10. Twenty **220 Ohm resistors** that can be used in circuits. We'll learn more about this later.
11. Various **LEDs** (Light Emitting Diodes). LEDs are very much like little light bulbs that work on DC (Direct Current). We'll learn more about this later. There are five each of red, green, yellow, blue, and RGB (red, green, blue). We'll learn more about these later.
12. Six **tactile switches** and five S8550 **PNP transistors**. We'll learn more about these later.
13. One **16GB MicroSD card** that comes preloaded with NOOBS that allows us to install the Raspbian operating system on the RPi.
14. Fifteen **6" male-male jumper wires** for use with the breadboard to prototype circuits.
15. Fifteen **3" male-male jumper wires** for use with the breadboard to prototype circuits.
16. Ten **3" female-female jumper wires** for use with the breadboard to prototype circuits.

While we're waiting, let's take a closer look at the RPi:



The RPi is a computer (just like a desktop or laptop), albeit a small one. It has the following features (starting at the top-right in the figure above):

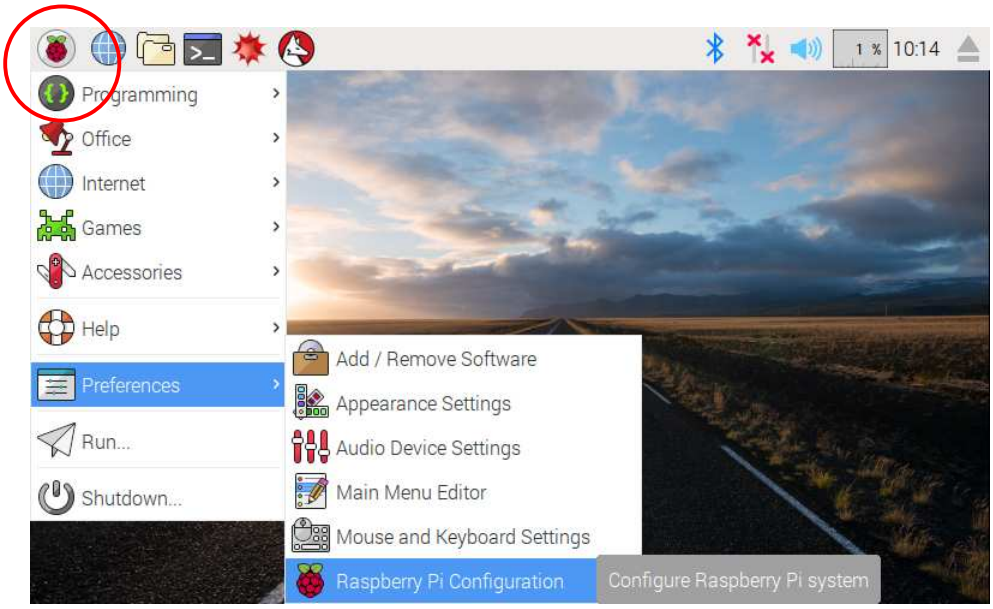
- Four **USB ports** for connecting peripheral USB devices (such as a keyboard or mouse);
- An **Ethernet port** for connecting an Ethernet cable (to provide wired Internet connectivity);
- A **3.5mm audio (and composite video) jack**, mainly for connecting speakers;
- A **CSI camera connector** to optionally connect the RPi camera;
- An **HDMI port** for connecting a display device (such as a monitor);
- A **MicroUSB port** for connecting the Raspberry Pi's power adapter;
- A **MicroSD card slot (**on the rear of the Raspberry Pi) for inserting a MicroSD card;
- A **DSI display connector** for connecting the touchscreen display;
- The **CPU** (Central Processing Unit) that serves as the heart of the Raspberry Pi; and
- A **GPIO header** that provides various General Purpose Input and Output capabilities (such as input sensors and output devices).

The RPi in the kit is a v3 Model B with a 1.2GHz 64-bit quad-core BCM2837 CPU, 1 GB of RAM (Random Access Memory), built-in WiFi and Bluetooth, and VideoCore IV graphics.

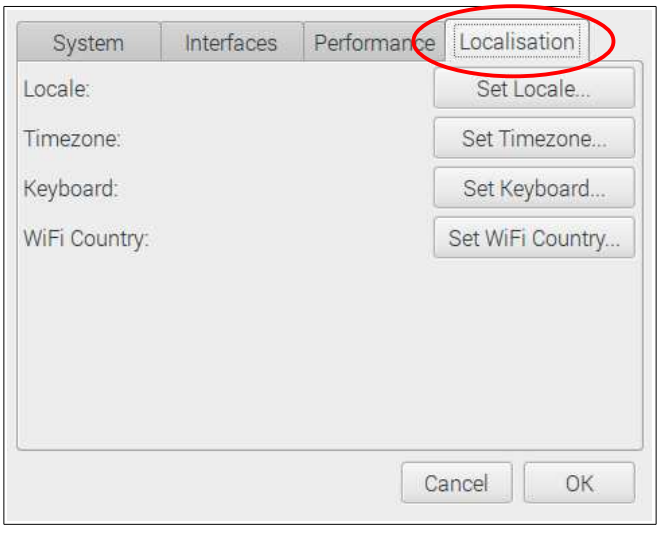
The GPIO interface and ribbon cable extends the RPi's GPIO pins to a breadboard, making it easier to prototype circuits. For example, LEDs, resistors, and other electronic components can be easily placed on the breadboard. Typically, the breadboard will be provided with power from the RPi through several GPIO pins. Sometimes, we may use input devices (such as push-button switches) that can be *read* by the RPi by our computer programs. We will prototype several circuits through various activities in the Living *with* Cyber curriculum.

Configuration

After the installation process is complete, the system reboots to the desktop. The next step is to properly configure the RPi by launching the RPi configuration tool via the **Raspberry Pi icon** at the upper-left of the Desktop (also known as the **start button**), followed by **Preferences and Raspberry Pi Configuration**:



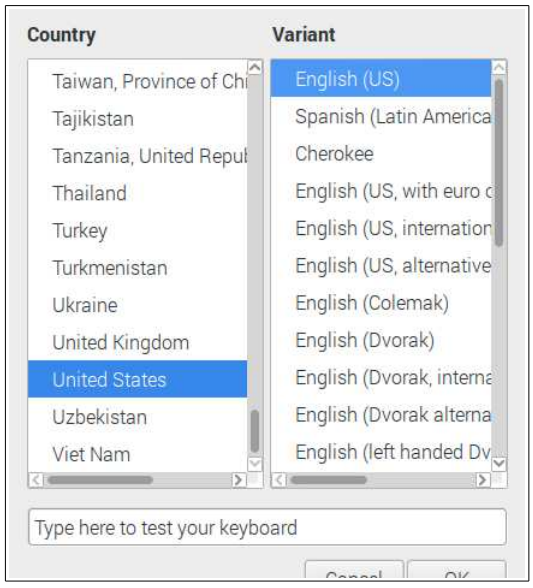
Select the **Localisation tab**:



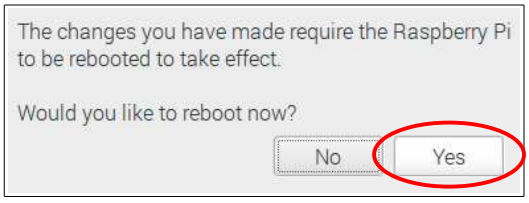
Click on the first setting, **Set Locale**, and select **US (USA)** as the country and **en (English)** as the language. When done, press **OK**.

Click on **Set Timezone**, then set the area to **CST6CDT** and click **OK**:

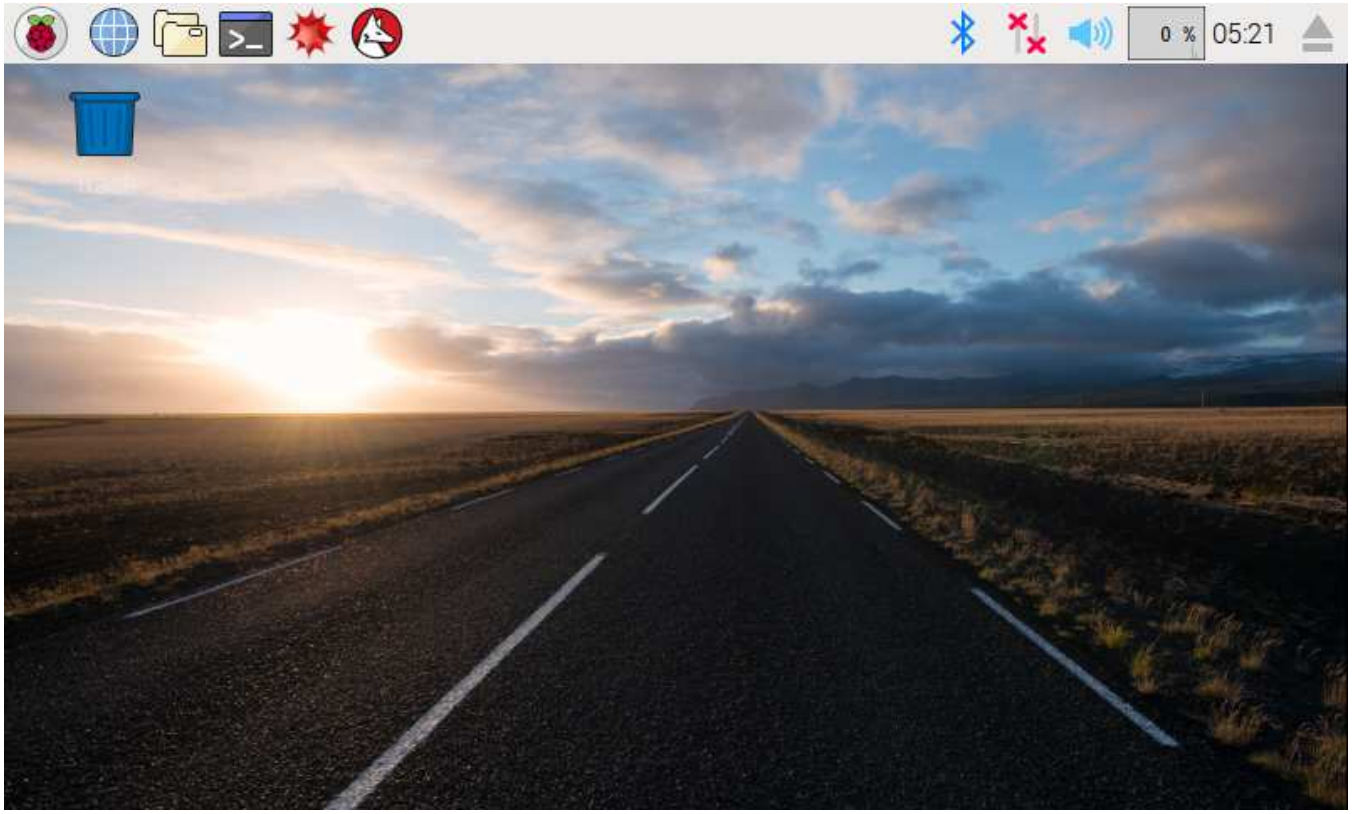
Click on **Set Keyboard**, then set the country to **United States** and variant to **English (US)** and click **OK** (note that the OK button is beneath the screen viewing area; therefore, you will need to move the Keyboard Layout window up a little):



Lastly, reboot the RPi by clicking on **Yes**:

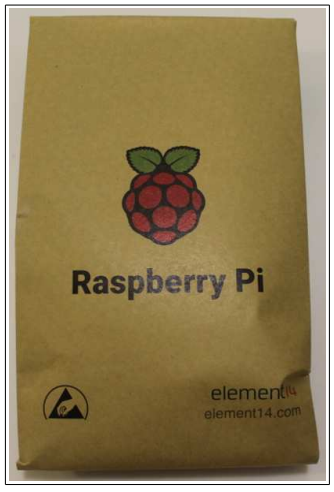


After the system reboots, you will be logged in to the RPi desktop with the updated localization settings. The **Wastebasket** is now the **Trash**:



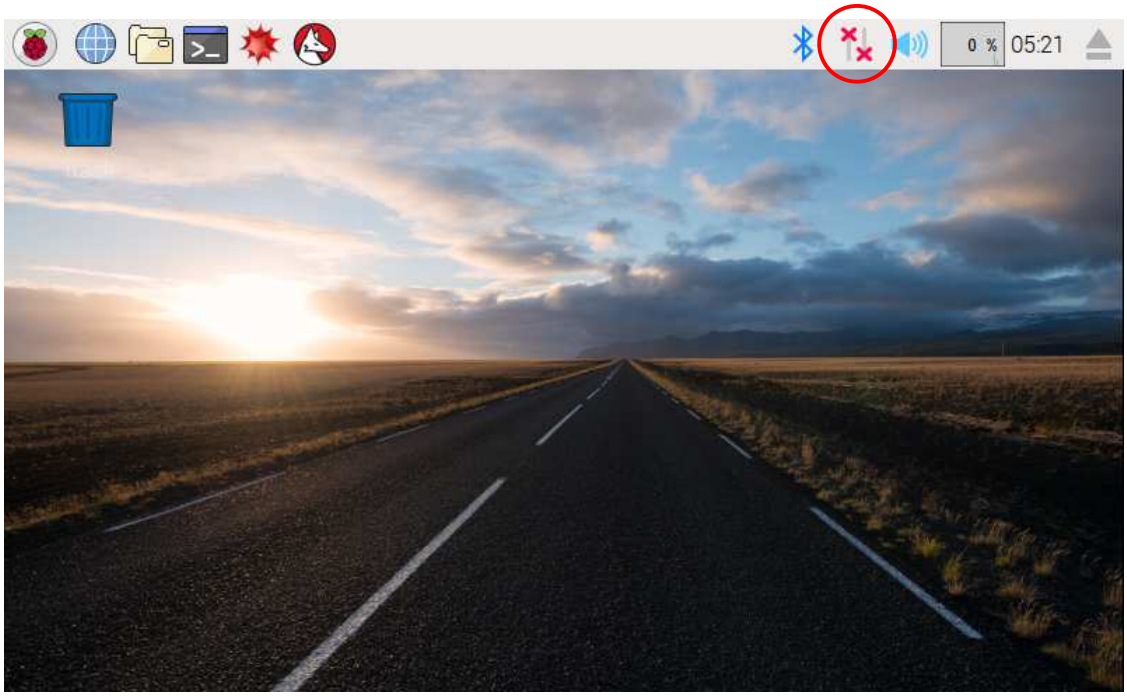
Congratulations! You have successfully configured your Raspberry Pi. Now we can start having fun!

A quick note: you can store all of the manuals and leftover items from the LCD touchscreen, Raspberry Pi, Smart Pi Touch stand, etc, in the paper bag that the LCD touchscreen was packaged in:



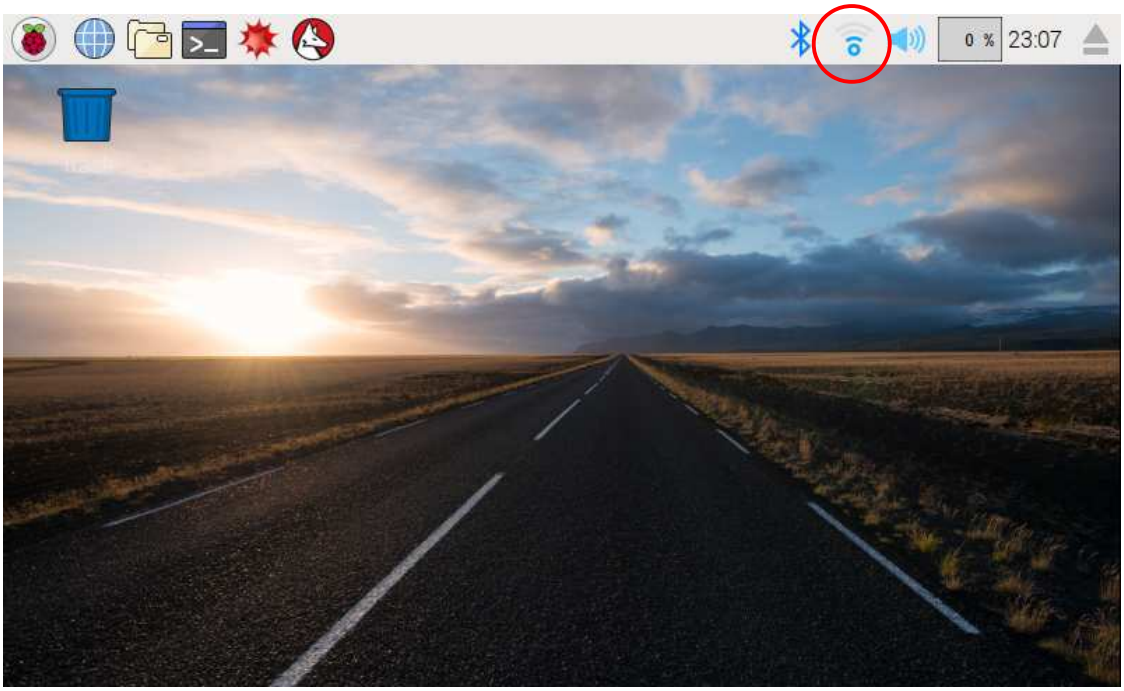
Exploration
You may notice that the desktop looks quite different than what you are probably normally used to. The Raspberry Pi is running an operating system called Raspbian which is a version of a broader operating system known as Linux. Many operating systems exist, some of which you may be familiar with: Windows, MacOS, Linux, Unix, Solaris, and so on. Although they may look different, they all do the same thing: allow you to operate the system.

The first task at this point is to get Wi-Fi working so that we can get on the Internet. In order to do that, click on the network connection icon at the top-right of the desktop and select the **cyberlab** WiFi network:

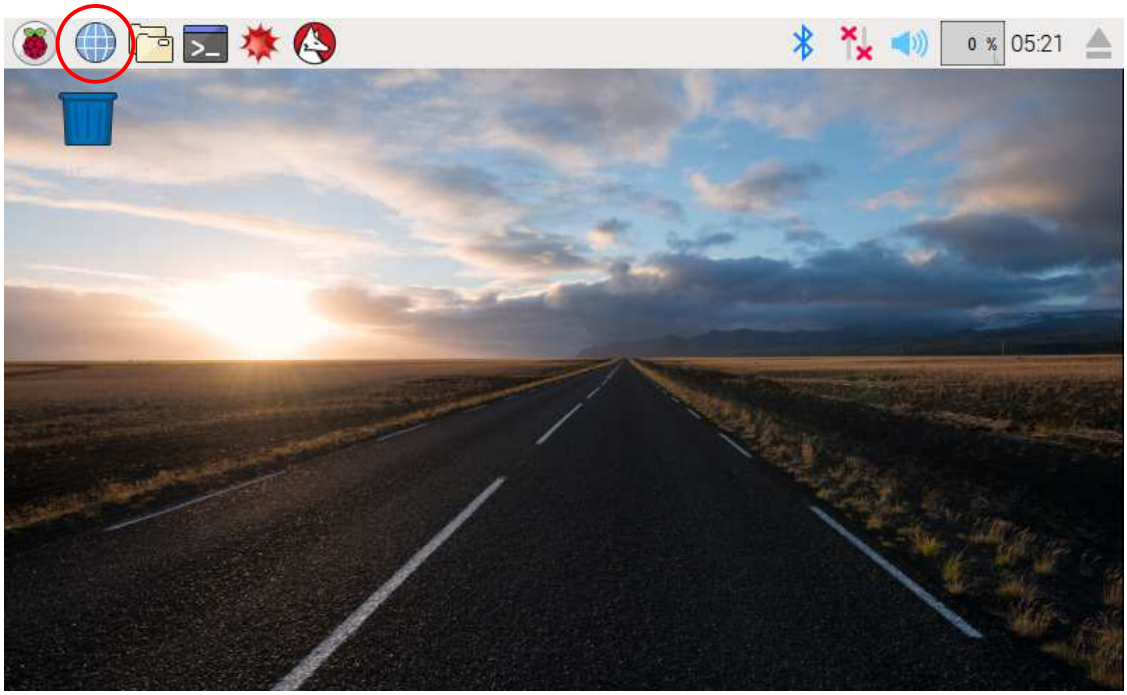


Since the WiFi network **cyberlab** is encrypted, it requires a passphrase to connect. Your prof will let you know the passphrase to type when prompted:

When you are connected, the network connection icon will change:

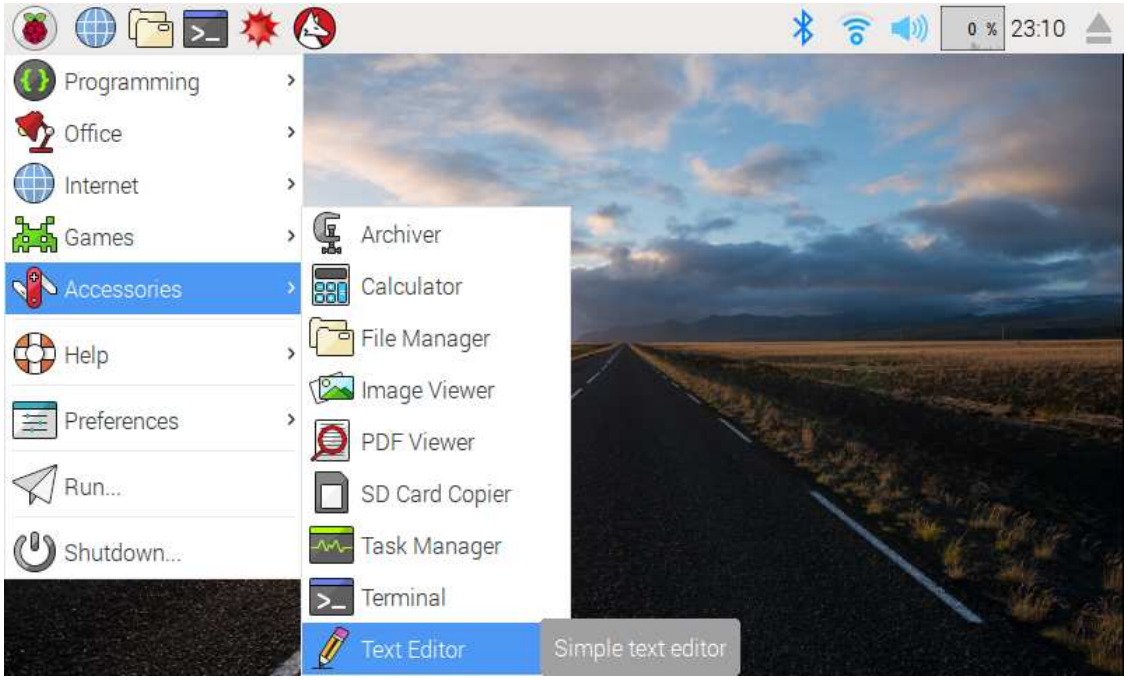


To test the network connection, let's browse to the Living *with* Cyber web page. To do this, we will need to open a web browser by clicking on its icon in the top left of the desktop:

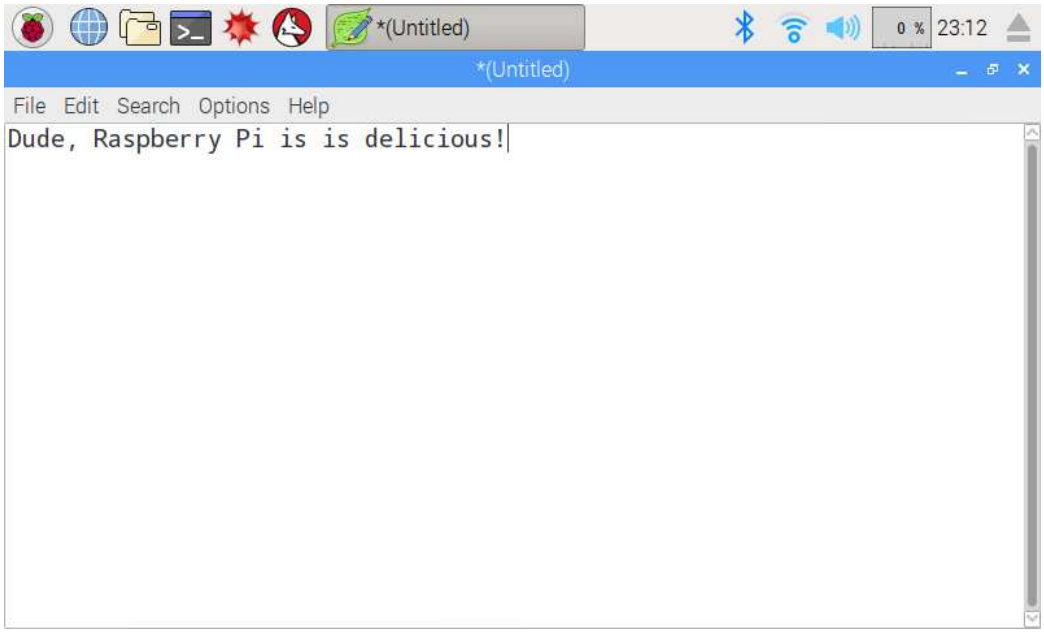


Once the web browser loads type **www.livingwithcyber.com** in the address bar at the top, then press **Enter**. If your Internet connection is working, it will take you to the Living *with* Cyber web site:

There are other interesting applications that we will make use of often. As an example, why don't we try to create a text document. Click on the start button, then on **Accessories** and **Text Editor**:

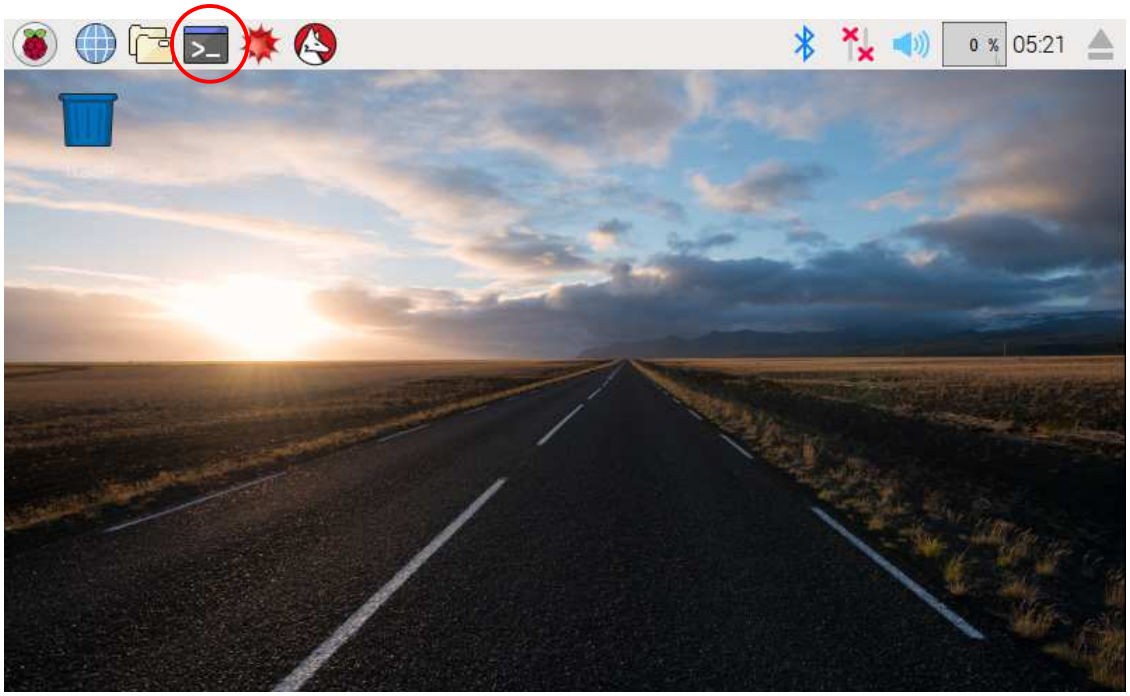


You can type anything you want. This application is similar to **notepad** on Windows systems:

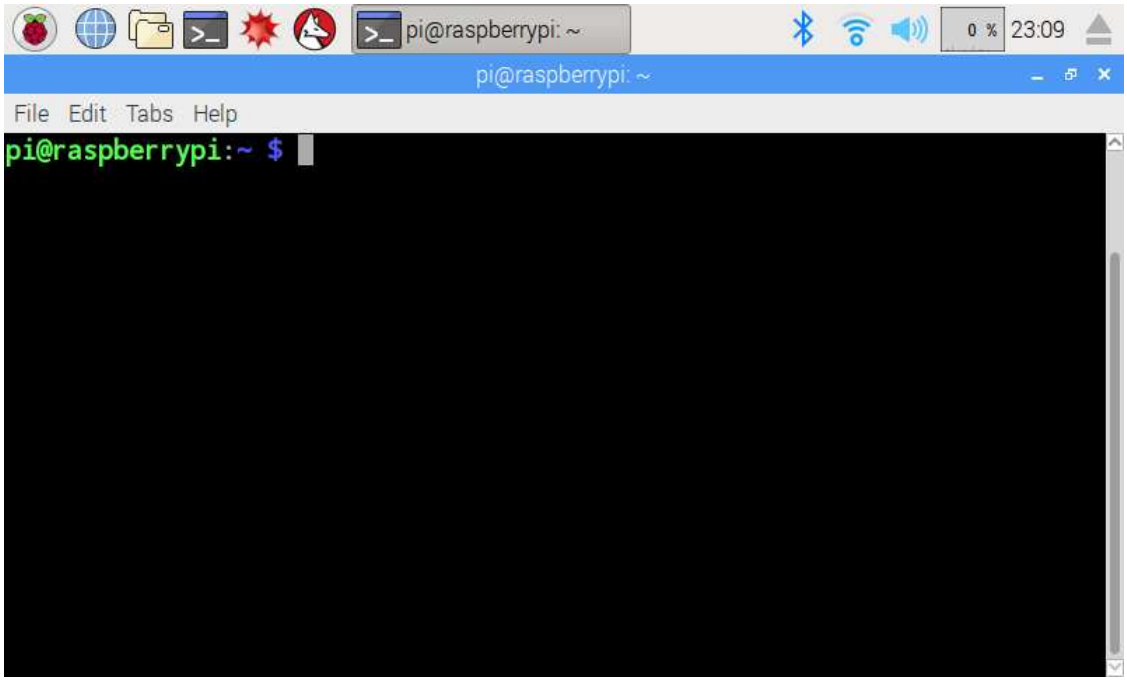


Close the text editor (but don't save the text file for now).

At the moment, you are interacting with the operating system on the Raspberry Pi by clicking around with your mouse. This kind of interface is known as a GUI (Graphical User Interface). There are other kinds of interfaces. One that is still widely used (particularly on the type of operating system that is installed on your Raspberry Pi) is a text-based interface called the **terminal**. It does exactly the same thing; however, instead of clicking around on icons and menu items, you type commands. Open a terminal by clicking on the monitor icon in the top left of the desktop:



Once the terminal loads, a rather simple text-based interface to the operating system is provided:

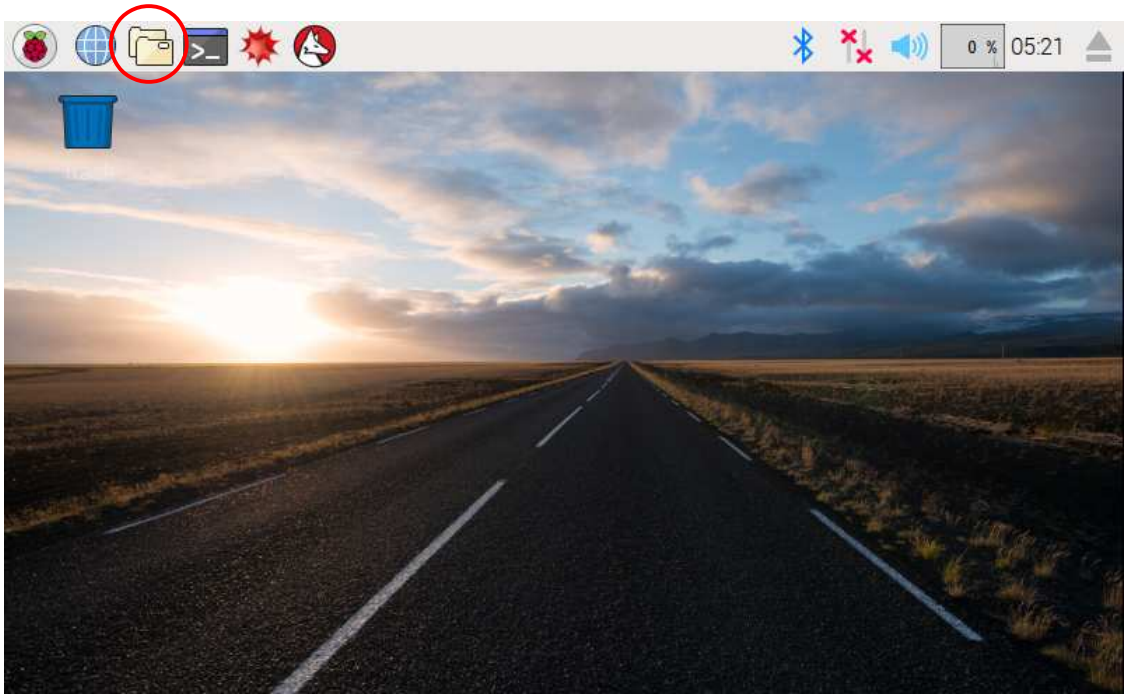


You will notice some text that is colored green and blue (by default) with a blinking cursor next to it. This is called a **prompt**, and it provides useful information. The prompt **`pi@raspberrypi:~ $`** can be broken down into several parts.

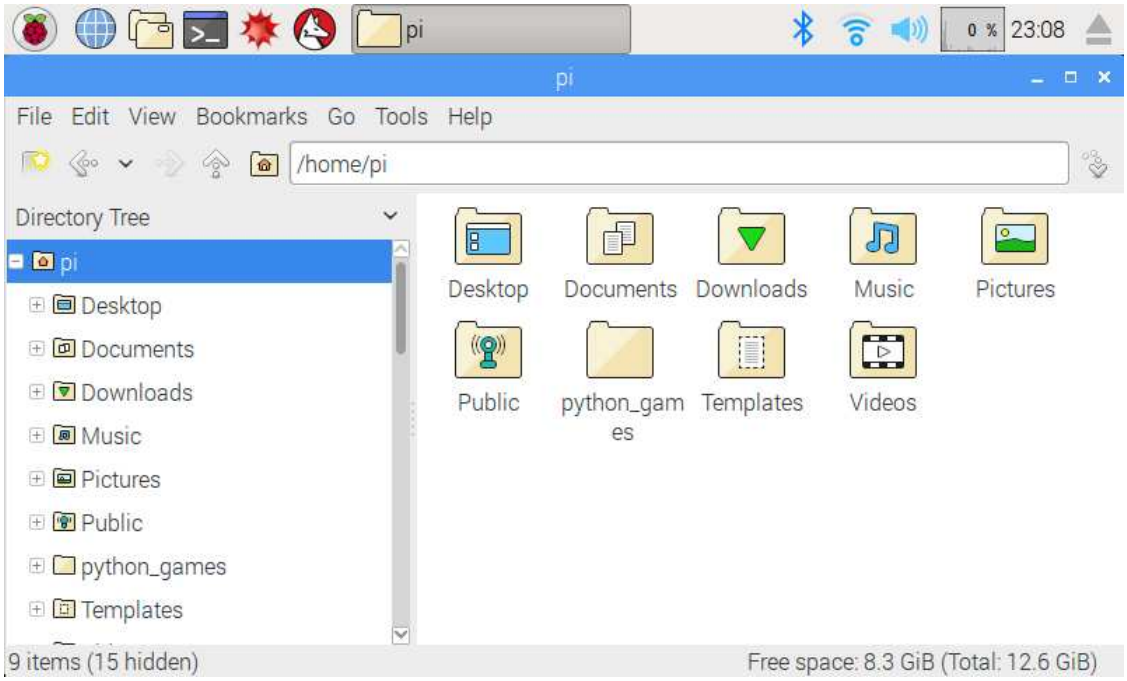
The first part, **`pi`**, represents the user that is currently logged in. That is, you are logged in to the operating system as the user pi. The second part, **`raspberrypi`**, represents the name of the system. So, the user pi is logged in to the system called raspberrypi; hence, pi@raspberrypi.

The third part, **`~`**, represents where you are on the system. You will learn that computers (like the Raspberry Pi) have space where files are stored. Think of a filing cabinet with folders that contain files. There can exist folders within other folders, and files within folders. We separate nested folders and files with a forward slash: /. So, for example, we could be in a folder called **homework** that is contained within another folder called **LwC**. The system would represent this as **/LwC/homework**. If we were looking at a file called **homework1** inside of the **homework** folder, the system would represent this as **/LwC/homework/homework1**. In our case, the system specifies ~, which means that we are at the user pi's home folder (where all of pi's documents are stored by default).

In fact, you can open a GUI-based file explorer by clicking on the file icon as shown below:

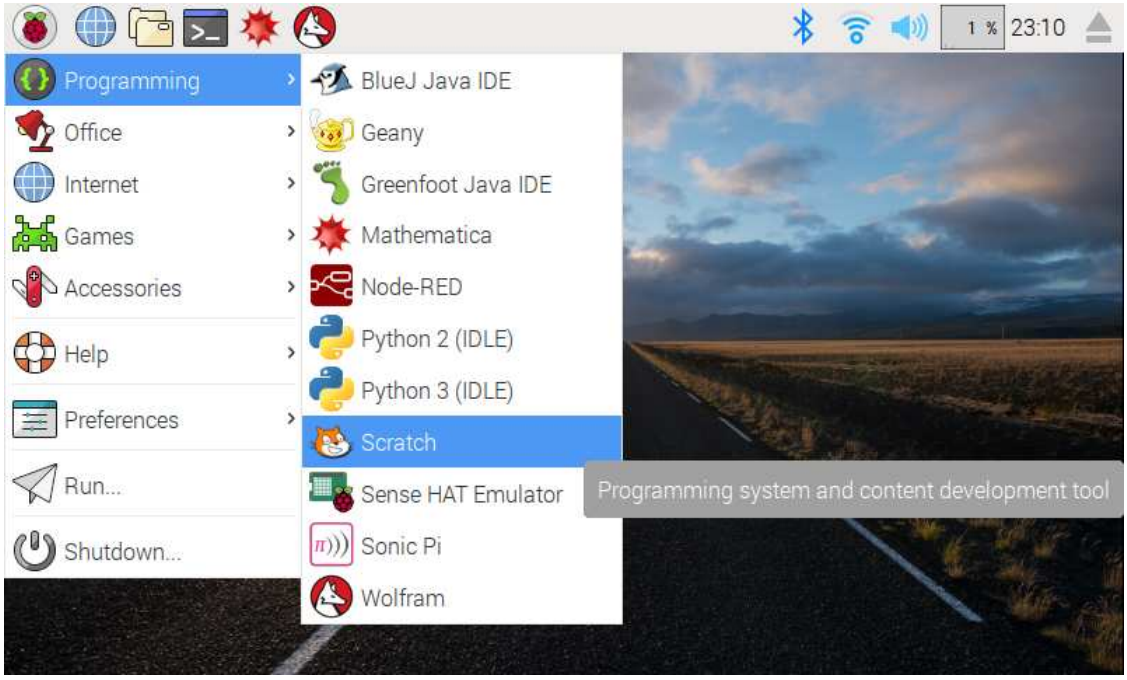


By default, you are in the logged in user's home folder. Since you are logged in as the user **pi**, then you are in that user's home folder. From there, you can explore the file system!

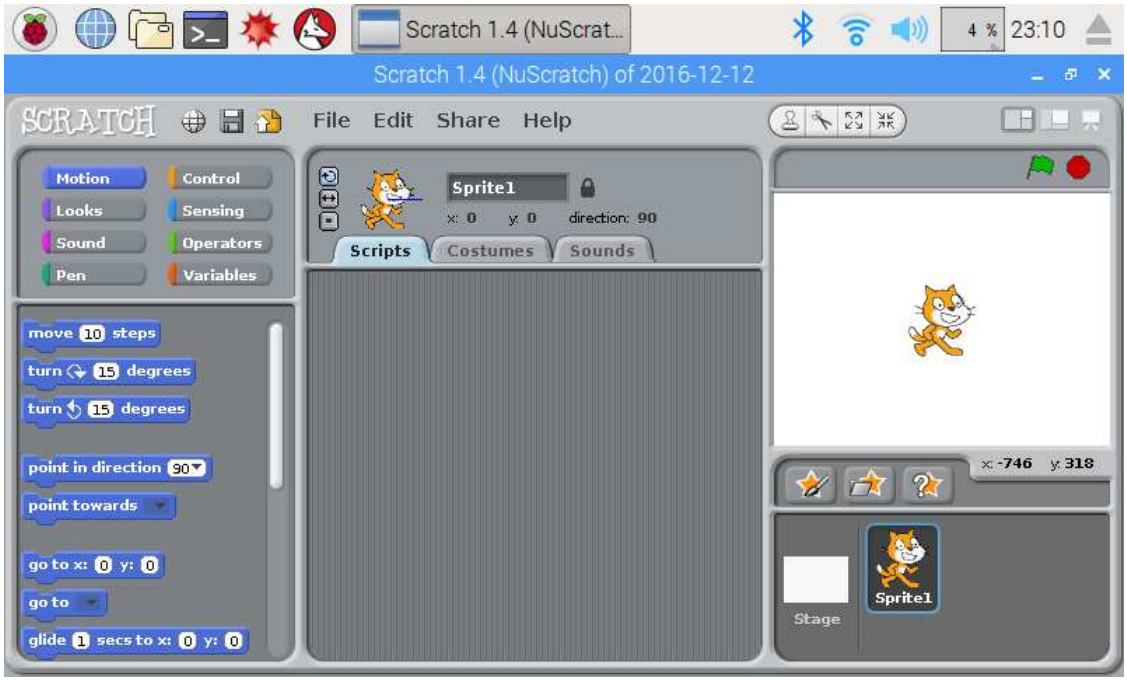


Scratch

In the next Raspberry Pi activity, you will learn about an educational programming language called Scratch. If there's time left in this activity, why don't you launch Scratch now and see if you can try to play around a bit! Launch Scratch by clicking the start button as follows:



Here is the Scratch interface:



Explore the interface (don't be afraid to try stuff). See if you can figure out how to make the cat move around the white area of the screen (called the stage).

Congratulations, you have finished your first Raspberry Pi activity!

Raspberry Pi Activity: *Scratch*ing the Surface

In this activity, you will learn about the Scratch programming language and design a simple game. You will need the following items:

- Raspberry Pi B v3 with power adapter;
- LCD touchscreen; and
- Keyboard and mouse.

The goal of this activity is to introduce you to the Scratch programming language and take you through the process of making a simple game in Scratch. Various programming constructs will be utilized and discussed (e.g., data types, constants, variables, sequence, selection, repetition, etc).

Although you won't actually design complex programs that solve interesting problems (yet), you will explore algorithm design and computer programming in Scratch, a visual programming language that replaces syntax with puzzle pieces. Unlike programming languages that are used in practice (e.g., C++, Java, Python), Scratch is intended for education and provides a great starting point for novice programmers. But don't get boxed in to the idea that Scratch is somehow not powerful. In fact, it is actually quite powerful and allows you to create games, animations, and interactive stories.

Scratch

Scratch is a basic programming language that utilizes puzzle pieces to represent the properties in the language (e.g., sequence, selection, and repetition). The programmer decides what pieces to use in order to implement the algorithm, and the puzzle pieces help identify what actions or statements can fit with each other and in what order they will be executed. More robust languages such as Python are entirely text-based where statements are text instructions used to represent actions.

In real physical puzzles, certain pieces have meaning and can only be used in certain places (e.g., edge pieces and corner pieces). This is similar in Scratch, in that certain pieces have meaning and must be used in certain places in our programs. Some of a puzzle's pieces can only be combined with certain other pieces so that they make sense.

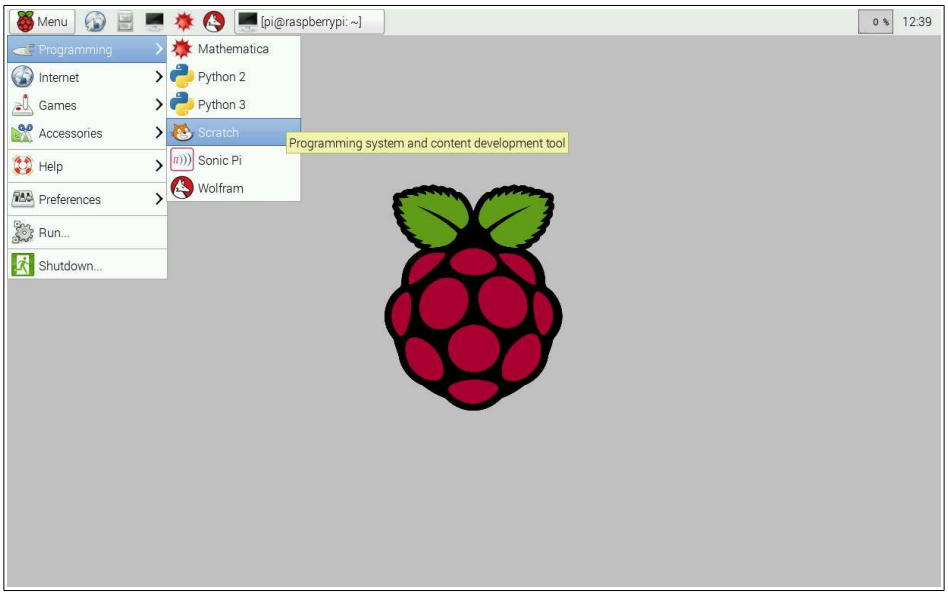
Scratch programs consist of scripts and is sprite driven. That is, a set of scripts can be defined for each sprite in a Scratch program (which we can more appropriately call a project). Scripts that are executed when the green flag is clicked can be defined for each sprite, and these scripts will execute *simultaneously* for each sprite! Sprites can communicate by way of broadcasting messages that can be received by other sprites. This is, in a way, a characteristic of object-oriented programming languages, where objects can communicate with each other by sending messages.

Scratch scripts are made up of various puzzle pieces (or blocks) that serve various functions. Blocks in the *motion* group provide programming constructs that deal with the movement or placement of sprites, while blocks in the *looks* group control anything related to the appearance of sprites (e.g., costume, graphical effect). *Sound* blocks provide the ability to incorporate sound in our Scratch programs, and *pen* blocks allow us to draw on the stage. *Control* blocks provide some of the most powerful functionality in Scratch. They allow us to implement selection and repetition quite easily, and in a variety of useful ways. They also provide ways to allow communication among sprites and to specify

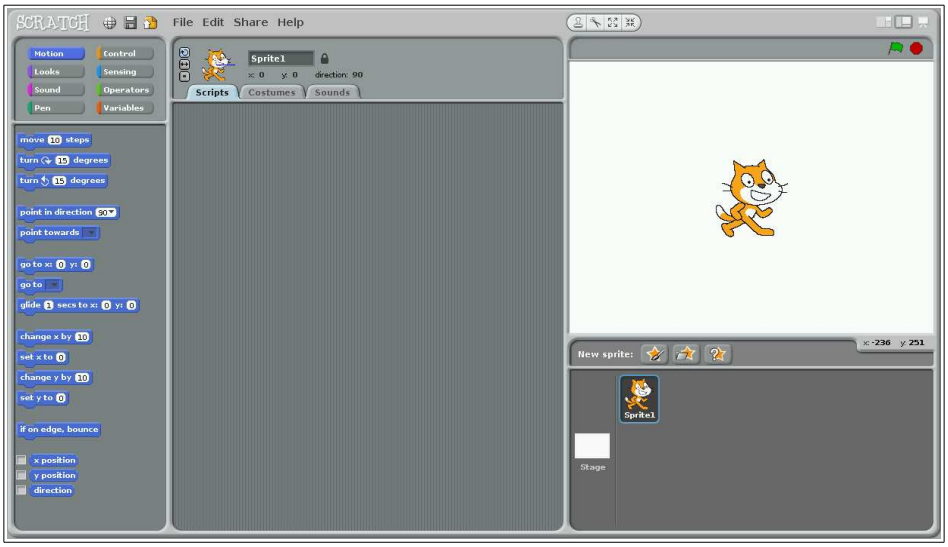
scripts to perform when events occur (e.g., when the green flag is clicked). Blocks in the *sensing* group provide ways of specifying input to our programs. We can, for example, detect if a sprite is touching another (and then specify some sort of action if desired). Blocks in the *operators* group provide math and string capabilities, something quite useful in our programs. These blocks allow us to compare values in order to determine an action to perform. Lastly, blocks in the *variables* group permit us to define variables and lists (i.e., a group of values). This is useful in virtually all programs, and you will find that declaring variables will become pretty routine.

Starting Scratch

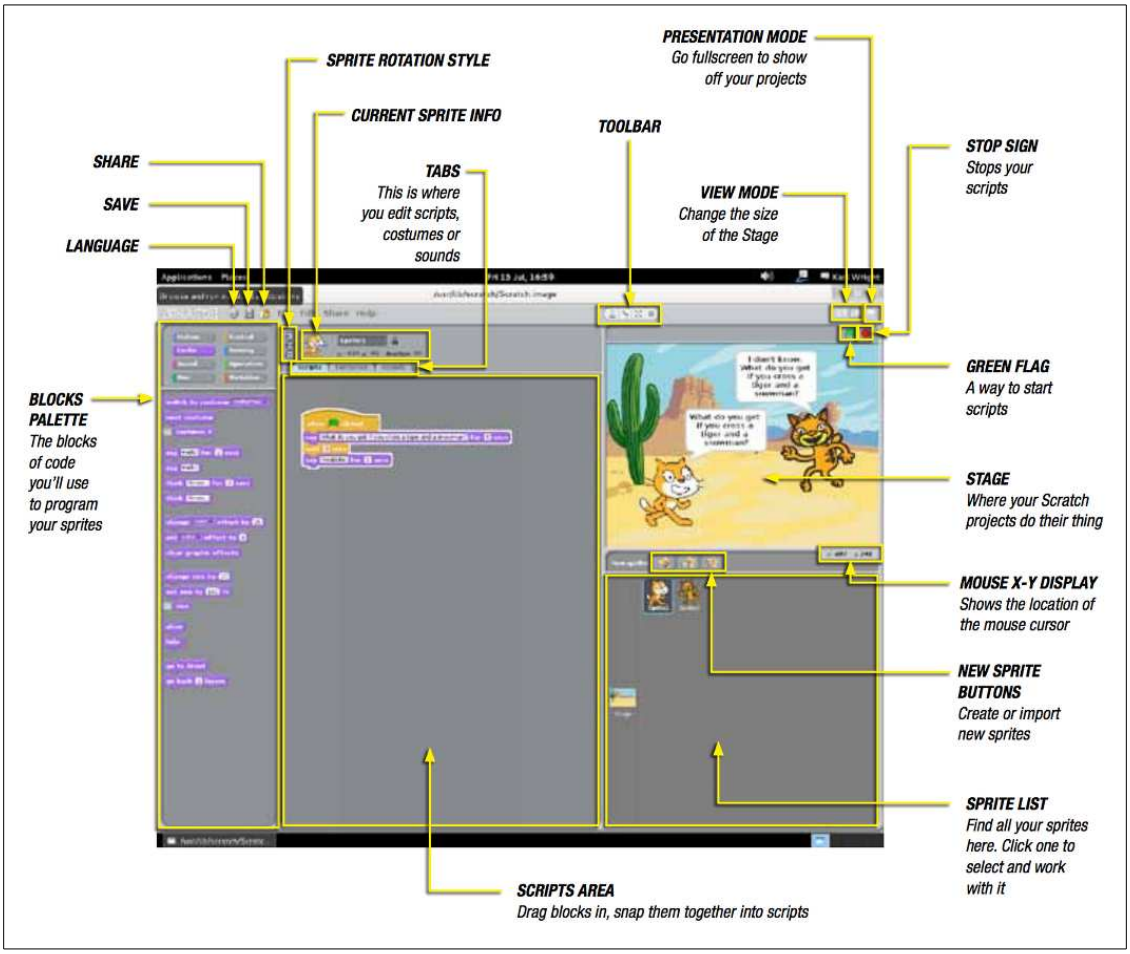
To begin, let's start Scratch from the menu:



After a short while, the Scratch interface will be displayed:



Although the interface looks a bit complicated, it really isn't. We won't make use of everything at first; however, here is an overview of the interface components:



The Scratch interface is quite busy; however, there are three main areas that you will find yourself interacting with often: the blocks palette, the scripts area, and the stage.

The **blocks palette** panel provides a variety of useful constructs that allow you to write programs. In total, there are eight different block types, accessible by clicking the groups in the top-left of the interface:

- *Motion*: anything related to the movement or placement of sprites (graphics);
- *Looks*: anything related to the appearance of sprites;
- *Sound*: anything related to incorporating sound in your programs;
- *Pen*: anything related to the pen which allows drawing on the canvas (background);
- *Control*: anything related to controlling the flow of programs;
- *Sensing*: anything related to detecting things (like movement, collisions, etc);
- *Operators*: anything related to math functions and string handling; and
- *Variables*: anything related to the declaration and upkeep of variables.

The **scripts area** is where you define your computer programs. This is done by dragging various blocks from the blocks palette and connecting them to make a program. It's almost like solving a jigsaw puzzle.

In fact, blocks have different types of notches and ridges that allow them to match up only to certain other blocks. This helps simplify the design of programs.

The **stage** is where your programs are executed. It's where to look to see if your code works...or not. On the stage, we can place sprites (graphics), variables, text, and drawings. At the top-right of the stage, a green flag and a red stop sign are used to start and stop your programs. The stage implements a two-dimensional coordinate system, where *x* and *y* represent the horizontal and vertical axes respectively. On our system with the LCD touchscreen, the center of the stage is at the point (0,0); the top-left corner is at (-240,180); the bottom-right corner is at (240,-180). Note that the stage is actually much larger (i.e., the cat sprite could technically be moved out of the viewable area of the stage).

Your first program

Let's create a simple program. Your task will be to move the cat sprite on the canvas. One useful block in the control blocks group is the **when green flag clicked** block. It is used to specify what to do when the green flag at the top of the stage is clicked (in other words, what to do when your program starts). We can add it to the scripts area by dragging it from the control blocks group in the blocks palette. Let's also add the first instruction to **move 10 steps** (pixels) in the direction the cat is facing (i.e., to the right). For this, we can utilize a block in the motion blocks group. Drag the move steps block to the scripts area until it snaps in place beneath the green flag block:

Click on the green flag in the top-right corner of the stage to run this program. You will notice that the cat moves very quickly a very small distance to the right. Click the green flag several times so that the cat repeatedly moves to the right a bit more.

Modify the program to move a different number of steps by changing the value in the move 10 steps block to something like -100 (which will move the sprite 100 pixels in the opposite direction that the cat is facing). Click the field that specifies the number of steps and replace 10 with -100:

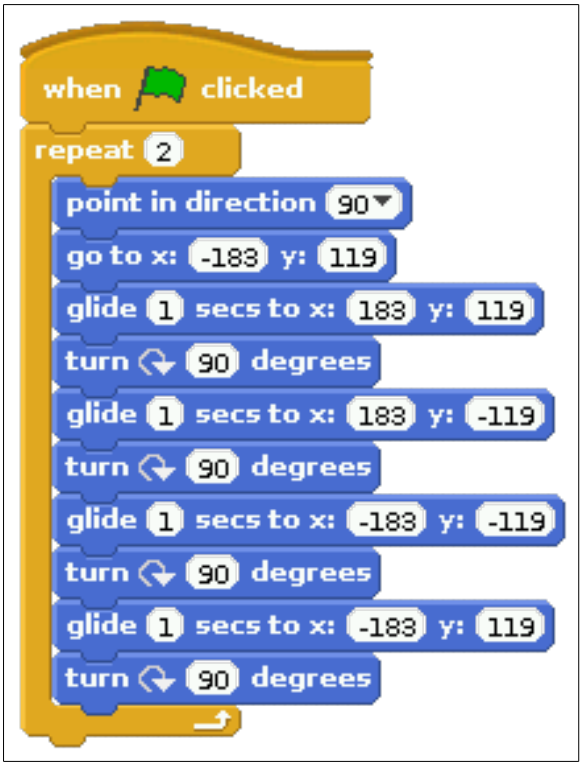
Clicking the green flag a few times moves the cat to the left quickly. This movement is too quick and quite joggy. In the motion blocks group, a **glide 5 secs to x,y** block allows motion to be more specifically defined. Let's replace the move steps block with this new block. To remove the move steps block, drag it away from the green flag block (you can just put it to the side if you anticipate using it again, or drag it back to the blocks palette to *trash* it). Tweak the values as you wish in the new motion block and watch the cat move smoothly to the specified coordinates:

You can actually click on the cat sprite to move it anywhere on the stage; then try running your program again.

An improvement?
Let's combine blocks to form a more complicated program. Create the following program and run it:

You'll notice that the program moves the cat around the perimeter of the stage, pointing in the direction of travel as it does so. How could the program be modified to repeat this some number of times (like 2)? There is a **repeat** block in the control blocks group that can be used to repeat an action some number of times.

Modify your program as follows:



Some of you will notice that the first two motion blocks (**point in direction 90** and **go to x,y**) can be moved out of and above the repeat block. Try it. You probably won't notice much difference, but these two blocks serve to initially orient and position the cat sprite. This really only needs to be done once at the beginning of the program.

At this point, let's cover some of the basic features of programming languages before continuing with the game.

Data types, constants, and variables

The kinds of values that can be expressed in a programming language are known as its **data types**. Scratch supports only two data types: text and numbers. The text data type provides the ability to represent non-numeric data such as names, addresses, English phrases, etc. The numeric data type allows the language to manipulate numbers, both positive and negative, whole numbers and fractions.

A **constant** is defined as a value of a particular type that does not change over time. Both numbers and text may be expressed as constants in Scratch. **Numeric constants** are composed of the digits 0 through 9 and, optionally, a negative sign (for negative numbers), and a decimal point (for floating point numbers). Numeric constants may not contain commas, dollar signs, or any other special symbols. The following are valid Scratch numeric constants: +15, -150, 15.01, 3200.

A **text constant** consists of a sequence of characters (also known as a string of characters – or just a **string**). The following are examples of valid string constants:

"She turned me into a newt."

"I got better."
"Very small rocks."

Note that the quotes surrounding the strings are not actually necessary to define a text constant in Scratch.

A **variable** is defined to be a named object that can store a value of a particular type. Scratch supports two types of variables: text variables and numeric variables. Before a variable can be used, its name must be declared. Variables are declared in Scratch through the *variables* blocks group.

Input and output statements
In order for a computer program to perform any useful work, it must be able to communicate with the outside world. The process of communicating with the outside world is known as input/output (or I/O). Scratch includes various input and output statements, although they are not implemented in the same way as other *real* programming languages such as Python or Java.

For example, in Scratch, individual sprites can **say *something* for n secs** or **think *something* for n secs**, displaying voice or thought bubbles with text. These are located in the *looks* blocks group. Sprites can also switch costumes, and programs can play sounds, draw with the pen, and so on. These are all output statements. Input statements include sensing when sprites are touching (or near) other sprites, or at the edge of the stage. Scratch can also ask the user for input (either text or numeric), and store this input to a variable. Many input statements in Scratch are located in the *sensing* blocks group. Most imperative languages include mechanisms for performing other kinds of I/O such as detecting where the mouse is pointing and accessing the contents of a disk drive.

The flexibility and power that input statements give programming languages cannot be overstated. Without them the only way to get a program to change its output would be to modify the program code itself, which is something that a typical user cannot be expected to do.

General-purpose programming languages allow human programmers to construct programs that do amazing things. When attempting to understand what a program does, however, it is vitally important to always keep in mind that the computer does not comprehend the meaning of the character strings it manipulates or the significance of the calculations it performs. Take, for example, the following simple Scratch program:

This program simply displays strings of characters, stores user input, and echoes that input back to the screen along with some additional character strings. The computer has no clue what the text string "Please enter your name." means. For all it cares, the string could have been "My hovercraft is full of eels." or "qwerty uiop asdf ghjkl;" (or any other text string for that matter). Its only concern is to copy the characters of the text string onto the display screen.

Only in the minds of human beings do the sequence of characters “Please enter your name.” take on meaning. If this seems odd, try to remember that comprehension does not even occur in the minds of all humans, only those who are capable of reading and understanding written English. A four year old, for example, would not know how to respond to this prompt because he or she would be unable to read it. This is so despite the fact that if you were to ask the child his or her name, he or she could immediately respond and perhaps even type it out on the keyboard for you.

Consider this Scratch program:

Here, input is numeric instead of text. The program prompts the user for two numbers, which it then computes the sum for and displays to the user. Note that two variables were declared in the *variables* blocks group: `num1` and `num2`. The first number is captured and stored in the variable `num1`. The second number is captured and stored in the variable `num2`. What do you think would happen if the user did not provide numeric input and, for example, inputted “Bob” for the first number? In the *real world*, programmers must create robust programs that examine user input in order to verify that it is of the proper type before processing that input. If the input is found to be in error, the program must take appropriate corrective action, such as rejecting the invalid input and requesting the user try again.

Primary control constructs

In order to create programs capable of solving more complex tasks we need to examine how the basic instructions we have studied can be organized into higher-level constructs. The vast majority of imperative programming languages support three types of control constructs which are used to group individual statements together and specify the conditions under which they will be executed. These control constructs are: sequence, selection, and repetition.

Sequence requires that the individual statements of a program be executed one after another, in the order that they appear in the program. Sequence is defined implicitly by the physical order of the statements. It does not require an explicit program structure. This is related to our previous discussion on **control flow**.

Selection constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed. Basically, selection allows a human programmer to include within a program one or more blocks of *optional* code along with some tests that the program can use to determine which one of the blocks to perform. Selection allows imperative programs to choose which

particular set of actions to perform, based on the conditions that exist at the time the construct is encountered during program execution.

Selection

Selection statements give imperative languages the ability to make choices based on the results of certain condition tests. These condition tests take the form of **Boolean expressions**, which are expressions that evaluate to either *true* or *false*. There are various types of Boolean expressions, but the types supported in Scratch are based on relational operators. **Relational operators** compare two expressions of like type to determine whether their values satisfy some criterion. The general form of all Boolean expressions supported in Scratch is:

expression relational_operator expression

For example:

Scratch includes three relational operators for comparing numeric expressions:

< meaning *less than*
= meaning *equal to*
> meaning *greater than*

For example, when `x` is 15 and `y` is 25, the expression `x > y` evaluates to false, since 15 is not greater than 25. Here are some additional examples of Boolean expressions that use these relational operators. These examples assume that the variable `x` holds 15 and `y` holds 25:

true `15 > 25 - 20`

true `15 + 10 = 25`

false `15 = 15 + 1`

Notice that the last expression always evaluates to *false* regardless of the value of `x`. This is because there is no possible value for `x` that will be equal to that value plus one. Another point illustrated by these examples is that relational operators have a lower precedence than mathematical operators. During expression evaluation, all multiplication, division, addition, and subtraction operations are performed before any relational operations.

The relational operators also work for text expressions as follows:

< meaning *precedes*
= meaning *equal to*
> meaning *follows*

Note that Scratch does not differentiate between uppercase and lowercase letters. That is, `A` is equal to `a`. Here are some examples, assuming that the variable `x` holds `Bob` and `y` holds `bobcat`:

false `Bob = bobcat`

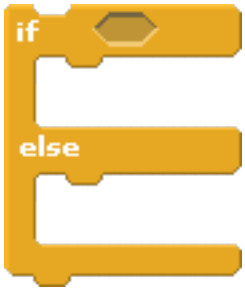
false `Bob > bobcat`

true `Bob < bobcat`

Since `Bob` precedes `bobcat` in alphabetical order, `Bob < bobcat` is *true*.

Selection statements use the results of Boolean expressions to choose which sequence of actions to perform next. Scratch supports two different selection statements: ***if-else*** and ***if***. An *if-else* statement allows a program to make a two-way choice based on the result of a Boolean expression.

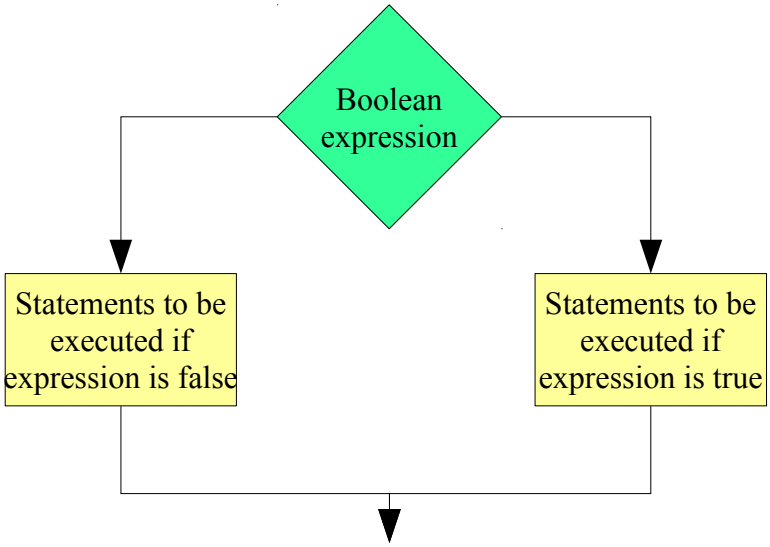
The general form of an *if-else* statements is shown below:



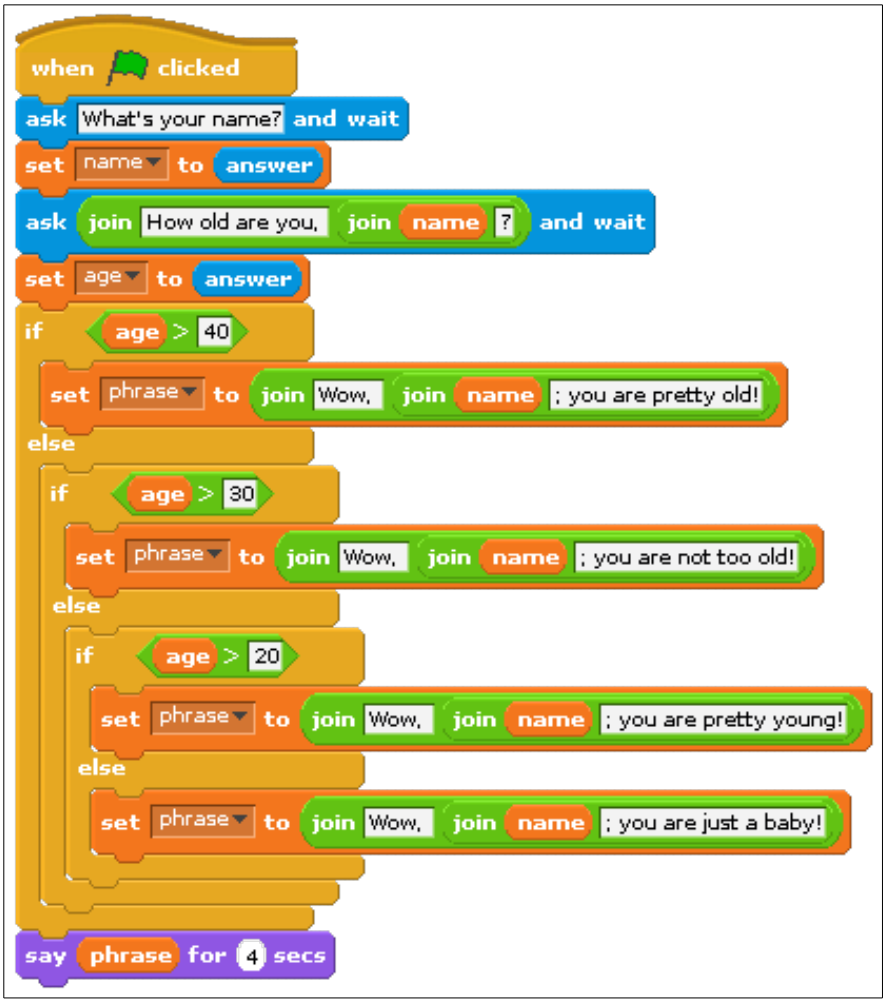
← statements to be executed if the expression is *true*

← statements to be executed if the expression is *false*

If-else statements specify a Boolean expression and two separate blocks of code: one that is to be executed if the expression is true, the other to be executed if the expression is false. Here's a flowchart for an *if-else* statement:



And here's an example of a program that implements an *if-else* statement (several, actually):



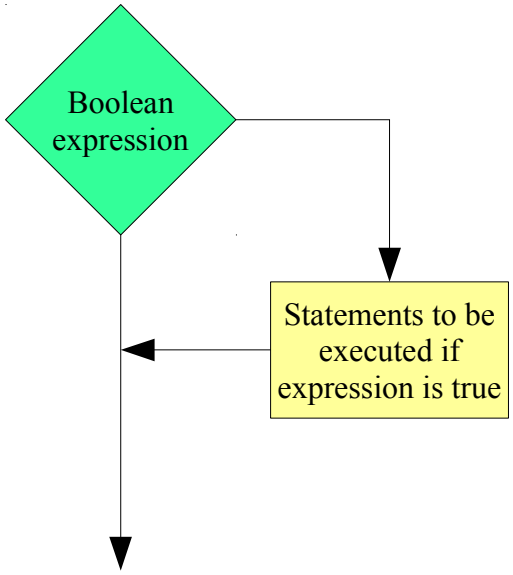
This program prompts the user to enter a name and age, and responds appropriately. It compares the user's age to several constants (40, 30, and 20), and sets the variable phrase depending on the user's age. If the user's age is more than 40, then the program responds that the user is pretty old; otherwise, the program then checks to see if the user's age is more than 30. If so, the program responds that the user is not too old; otherwise, it then checks to see if the user's age is more than 20. If so, it responds that the user is pretty young; otherwise (any age less than or equal to 20), it responds that the user is just a baby.

Note that we can nest an *if-else* statement inside of another *if-else* statement to provide more than two alternatives or paths. Here's the program above represented in pseudocode:

```
name ← prompt for name
age ← prompt for age
if age > 40
then
     phrase ← "Wow, " + name + "; you are pretty old!"
else
     if age > 30
     then
```

```
            phrase ← "Wow, " + name + "; you are not too old!"
        else
            if age > 20
            then
                phrase ← "Wow, " + name + "; you are pretty young!"
            else
                phrase ← "Wow, " + name + "; you are just a baby!"
            end
        end
    end
    display phrase
```

The *if* statement is similar to the *if-else* statement except that it does not include an *else* block. That is, it only specifies what to do if the Boolean expression is true. Here's a flowchart for an *if* statement:



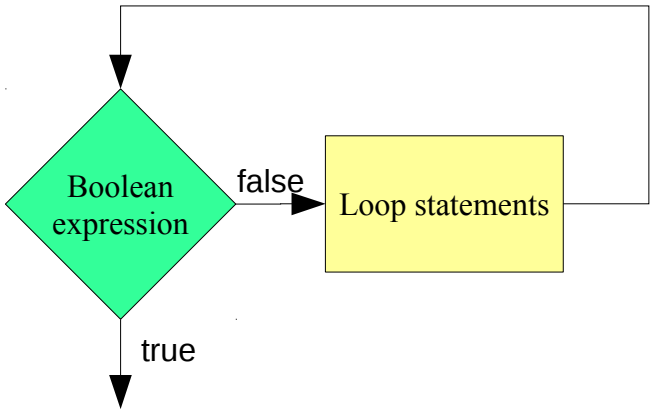
If statements are generally used by programmers to allow their programs to detect and handle conditions that require *special* or *additional* processing. This is in contrast to *if-else* statements, which can be viewed as selecting between two (or more) *equal* choices.

Repetition

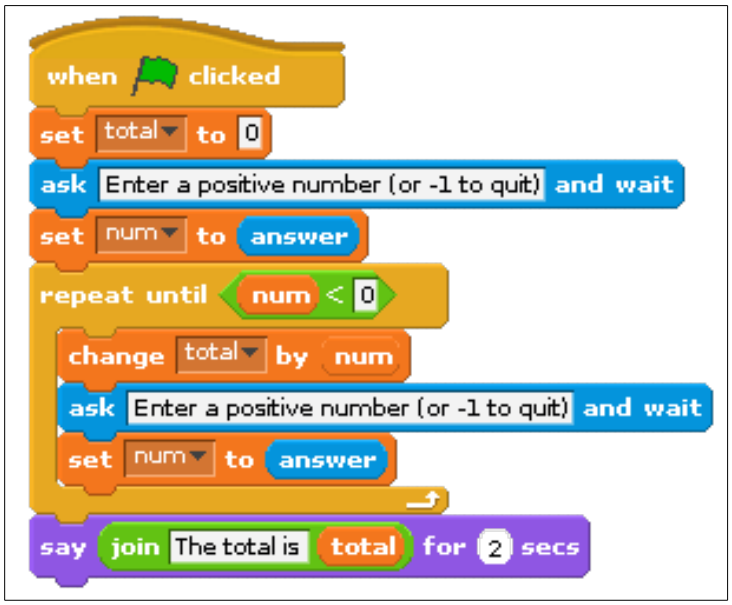
Repetition is the name given to the class of control constructs that allow computer programs to repeat a task over and over. This is useful, particularly when considering the idea of solving problems by decomposing them into repeatable steps. Repetition constructs contain exactly one block of statements together with a mechanism for repeating the statements within the block some number of times. There are two major types of repetition: iteration and recursion. **Iteration**, which is usually implemented directly in a programming language as an explicit program structure, often involves repeating a block of statements either (1) while some condition is true or (2) some fixed number of times. **Recursion** involves a subprogram (e.g., a function) that makes reference to itself. As with sequence, recursion does not normally have an explicit program construct associated with it.

Scratch supports iteration in two main forms: the *repeat* loop and the *forever* loop. The *repeat* loop has two forms: *repeat-until* and *repeat-n* (where *n* is some fixed or known number of times). The *repeat-until* loop is condition-based; that is, it executes the statements of the loop until a condition becomes true. The *repeat-n* loop is count-based; that is, it executes the statements of the loop *n* times.

Here's a flowchart of the *repeat-until* loop:



The Boolean expression is first evaluated. If it evaluates to false, the loop statements are executed; otherwise, the loop halts. Here is an example:

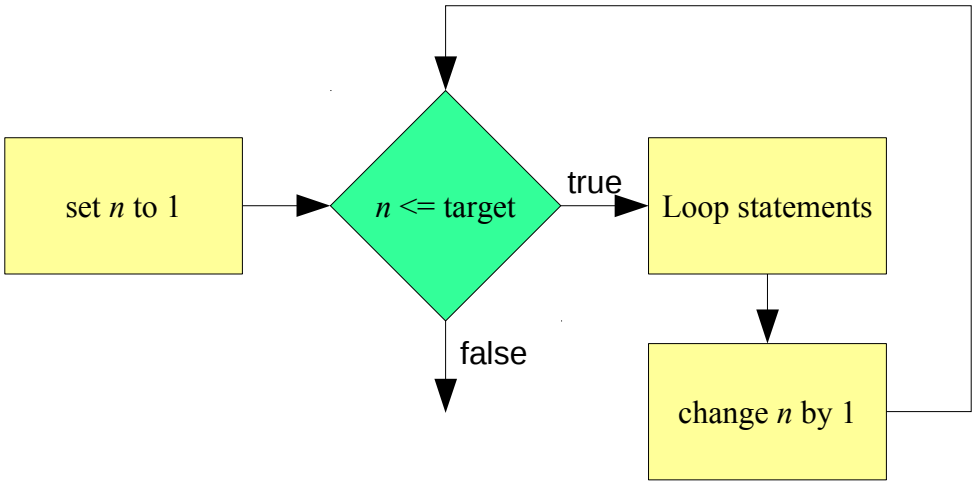


This program asks the user to enter a positive number or -1. If a positive number is entered, it is added to a running total. If -1 is entered, the program displays the total and halts. The *repeat-until* loop is used here to repeat the process of asking the user for input until the value entered is less than 0. It is interesting to note that although the program instructs users to enter -1 to quit, the condition that controls the loop is actually `num < 0` (which will be true for any negative number). Thus, the loop will actually terminate whenever the user enters any number less than zero (e.g., -5).

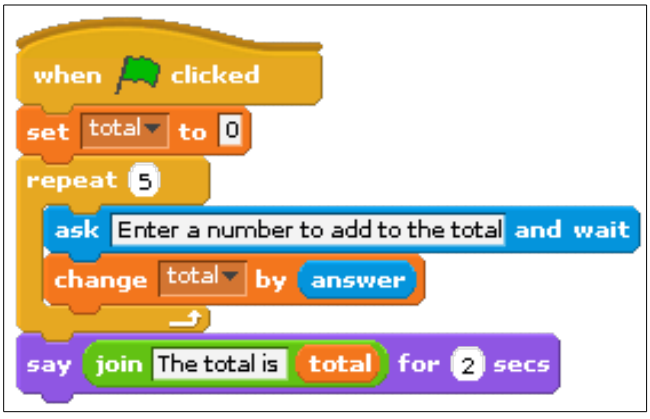
Here's the program above represented in pseudocode:

```
total ← 0
repeat
      num ← prompt for a positive number (or -1 to quit)
      if num > 0
      then
            total ← total + num
      end
until num < 0
display total
```

The *repeat-n* loop executes the loop statements a fixed (or known) number of times. Here's a flowchart of the *repeat-n* loop:



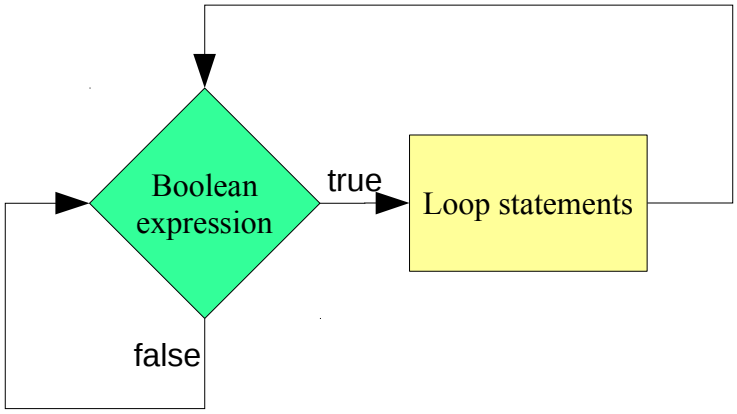
Although the programmer does not have access to a variable that counts the specified number of times (shown as *n* in the figure above), the process works in this manner. A counter is initially set to 1. A Boolean expression is then evaluated that checks to see if that counter is less than or equal to the target value (e.g., 10). If so, the loop statements execute. Once the loop statements have completed, the counter is incremented, and the expression is reevaluated. Here is an example:



This program asks the user to enter five numbers. Each time, the number is added to a running total. After all five numbers have been entered, the total is displayed.

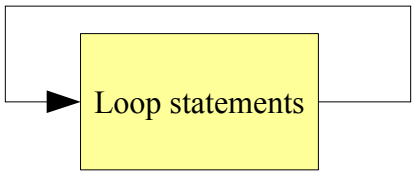
The forever loop also has two forms: *forever* and *forever-if*. The *forever* loop executes the statements of the loop forever. Well, it's not technically forever, since we can click the stop button to halt all scripts at any time. The *forever-if* loop is condition-based (like the *repeat-until* loop), and executes the statements of the loop if a condition is true. Note that the *forever-if* loop also runs forever (until the stop button is clicked). the difference is that the loop statements are only executed if the condition is true.

Here is a flowchart of the *forever-if* loop:



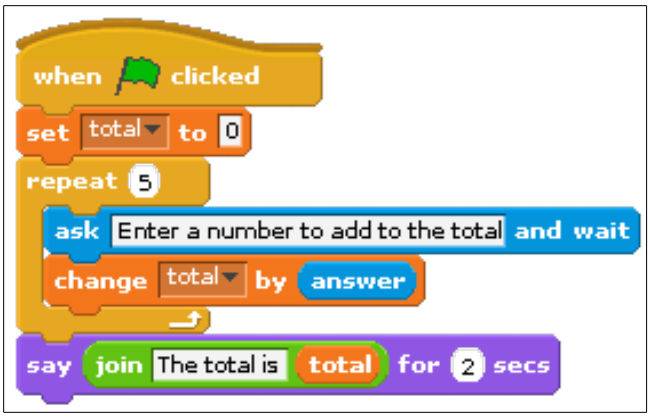
This loop construct is often used to perform real time checking of sprites and execute statements if, for example, the sprite is at a certain position on the stage. Another example is to constantly check the value of a variable that is changed by some other sprite. In this way, a sprite can monitor a variable, and when changed to an appropriate value, perform some action.

Lastly, here is a flowchart of the *forever* loop:

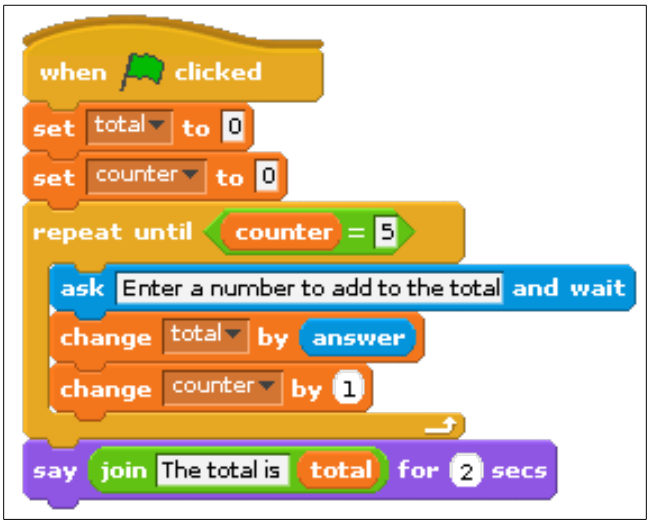


Pretty straightforward...

Note that any program written using a *repeat-n* loop can be rewritten as a *repeat-until* loop. Take, for example the *repeat-n* loop shown earlier:



Here it is, rewritten using a *repeat-until* loop:



The only difference is that, in the *repeat-until* loop, the programmer must keep track of (and modify) the counter each time the loop statements execute. In the *repeat-n* loop, Scratch automatically takes care of this.

Is the opposite true? That is, can every program that uses a *repeat-until* loop be rewritten using a *repeat-n* loop? The answer is no. A *repeat-n* loop simply loops a fixed or known number of times. A *repeat-until* loop repeats until some condition is true. That condition could be, for example, when the user inputs -1 to terminate. The idea of expressing a condition that represents a sentinel value in a manner that requires knowing how many times the loop statements will execute is nonsensical. There is no way to tell how many times the loop statements will execute until the user inputs -1. It could be the very first time (or the 10,000th). Because *repeat-n* loops can always be replaced with *repeat-until* loops, but not all *repeat-until* loops can be replaced with *repeat-n* loops, we say that the *repeat-until* loop is more general than the *repeat-n* loop.

Subprograms

A **subprogram** is a program within a program. Recall an earlier lesson on representing algorithms as to-do lists. One algorithm represented the steps necessary to *get to class*. One of those steps was *eat breakfast*. We noted how we could zoom in to that step and identify the sub-steps necessary to complete the *eat breakfast* step. Control flow shifted from the main to-do list to the *eat breakfast* to-do list when the *eat breakfast* step was encountered, and then returned to the main to-do list at the point where it left earlier. We can consider the *eat breakfast* to-do list as a subprogram.

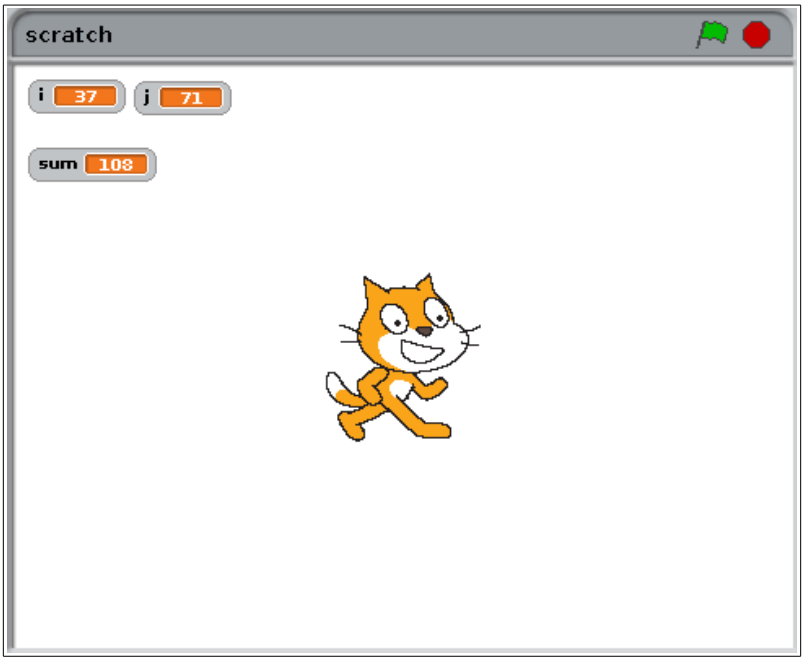
Very few *real* programs are written as one long piece of code. Instead, traditional imperative programs generally consist of large numbers of relatively simple subprograms that work together to accomplish some complex task. While it is theoretically possible to write large programs without the use of subprograms, as a practical matter any significant program must be decomposed into manageable pieces if humans are to write and maintain it.

When a subprogram is invoked, or called, from within a program, the *calling* program pauses temporarily so that the *called* subprogram can carry out its actions. Eventually, the called subprogram will complete its task and control will once again return to the *caller*. When this occurs, the calling program *wakes up* and resumes its execution from the point it was at when the call took place.

Subprograms can call other subprograms (including copies of themselves as we will see later). These subprograms can, in turn, call other subprograms. This chain of subprogram invocations can extend to an arbitrary depth as long as the *bottom* of the chain is eventually reached. It is necessary that infinite calling sequences be avoided, since each subprogram in the chain of subprogram invocations must eventually complete its task and return control to the program that called it.

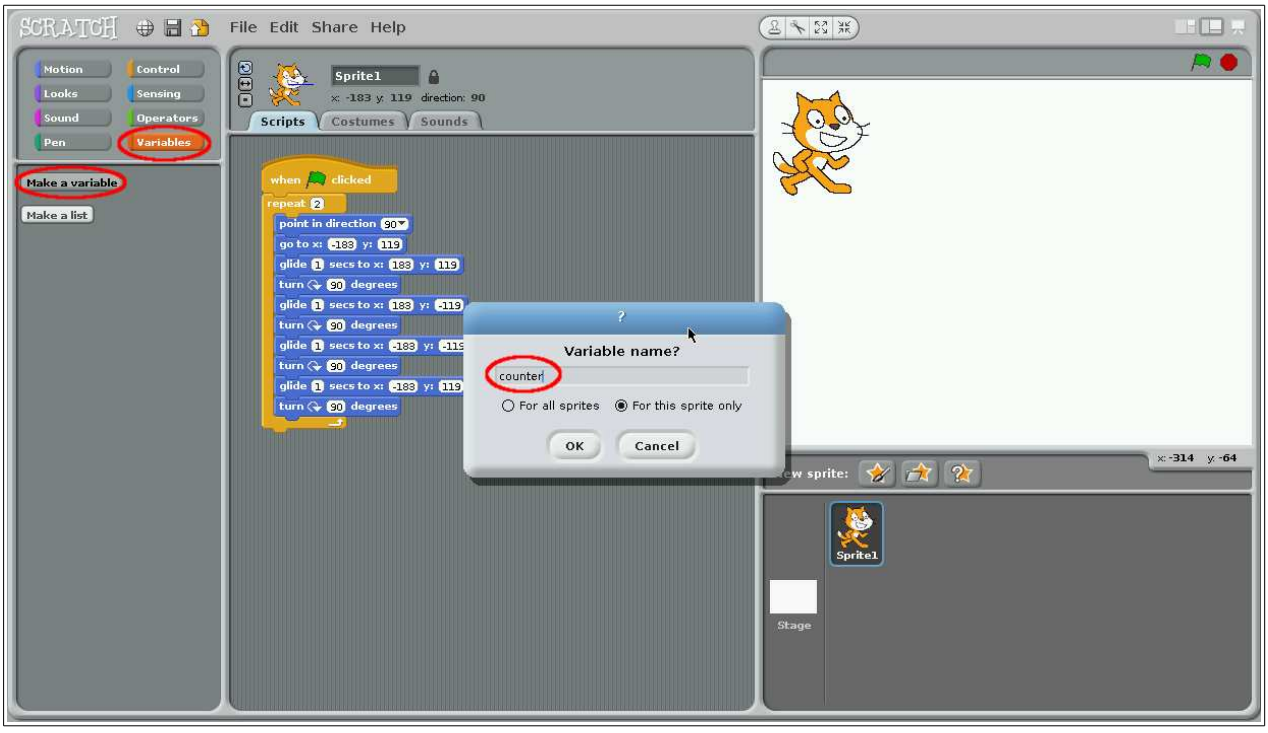
In Scratch, we define subprograms as broadcasts. That is, a sprite can broadcast a message that can be received by another sprite (or even the same sprite). We can think of this as calling a subprogram. When receiving a broadcast, we can specify the script (subprogram) associated with it. Here is a simple example of a subprogram that computes the sum of two numbers stored in the variables `i` and `j`:

The left side shows the subprogram defined by receiving the broadcast `addem`. The right side shows the broadcast (or the subprogram *call*). Assuming that the variables `i` and `j` have been declared and given numeric values, then the *addem* subprogram would add the two variables together and store the result to the variable `sum`. Here is an example when `i` is 37 and `j` is 71 (note that the *addem* subprogram has been *called* by being broadcasted):

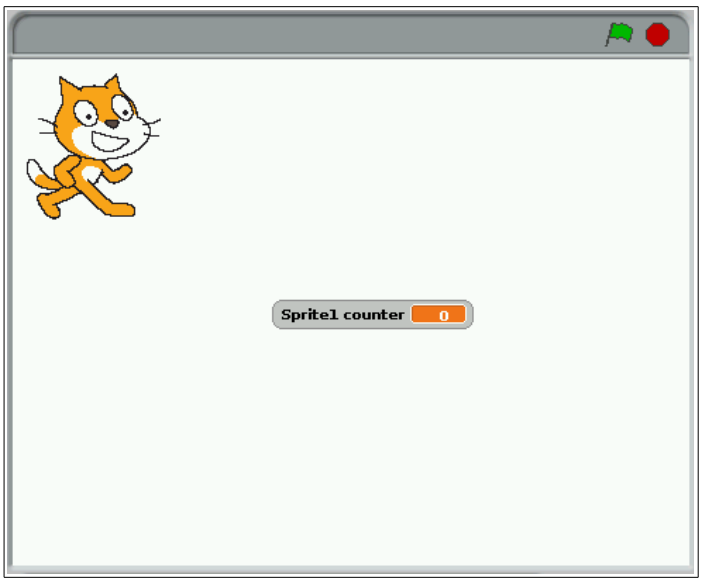


Back to the game

It would be neat to count the number of 90 degree turns that the cat makes during its journey. To do this, let's define a variable (called `counter`) that will be updated each time the cat turns. Defining variables can be done by selecting **make a variable** in the variables blocks group.

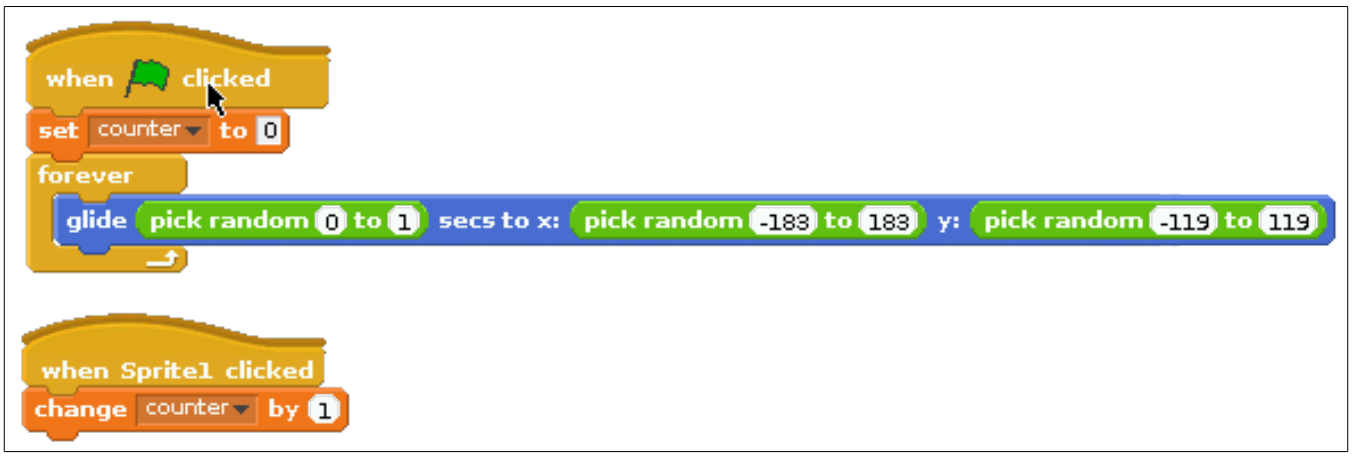


This adds the variable on the stage. Drag it to the center of the stage so that it doesn't get in the cat's way as it moves around:



To use the counter in your program, it will first need to be initialized (with the value 0) and then incremented each time the cat makes a 90 degree turn. We can modify our program as follows:

A first game?
Modify your program (you can save the current version first if you wish) so that it looks like this:

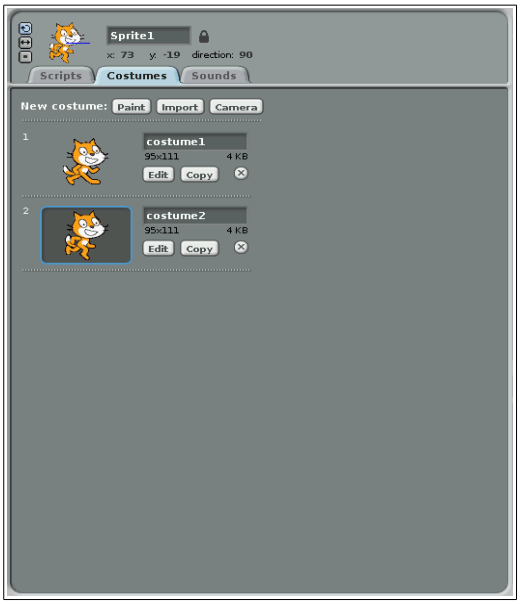


Notice that we now have two block groups in the scripts area. One that is executed when the green flag is clicked; and another that is executed when the cat sprite (Sprite1) is clicked. What does the program do?

Did you know?

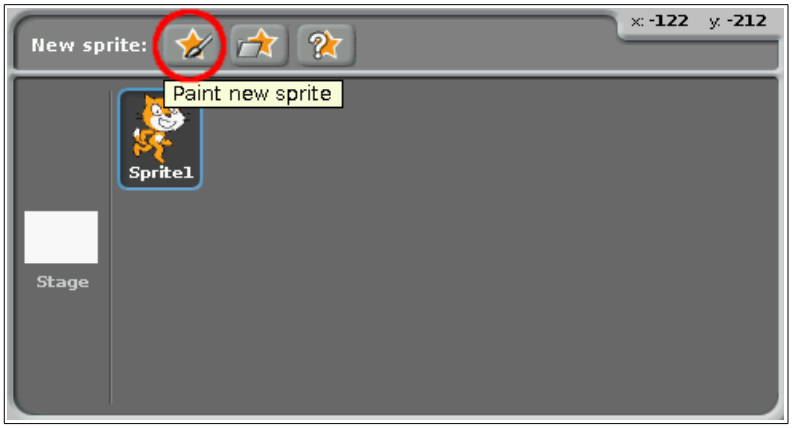
You can change the name of the cat sprite at the top of the scripts area. Make sure that the cat is selected in the **sprite list** below the stage.

Playing with sprites
At the top of the scripts area, there are several other tabs that provide sprite costume and sound tools. Click on the **costumes** tab. You will notice that the cat actually has two sprites, one named costume1 and another named costume2:



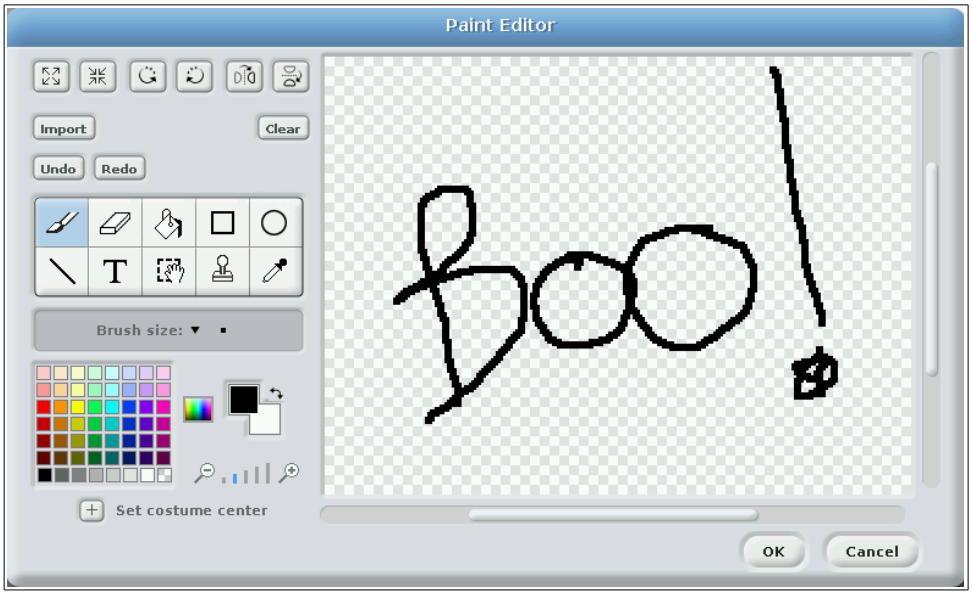
By alternating them, we can make it look as if the cat is walking or running. Modify your program as follows:

You can also create your own sprites via the paint new sprite button in the sprite list:



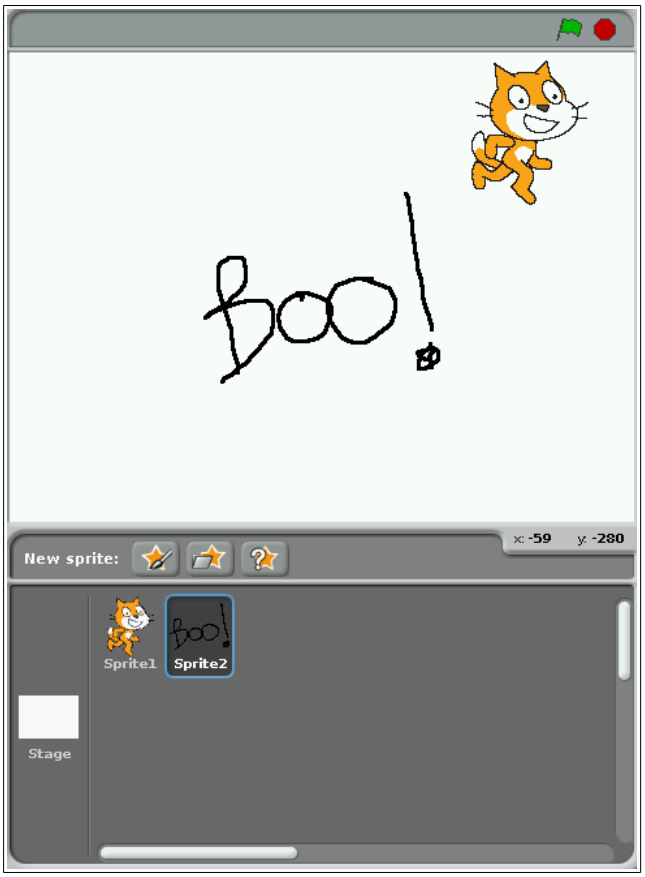
This displays a **paint editor** that can be used to create a new sprite of your design. The other buttons to the right allow saved sprites to be loaded (Scratch comes with many different sprites) and a random sprite to be added to the sprite list.

Try creating one now:



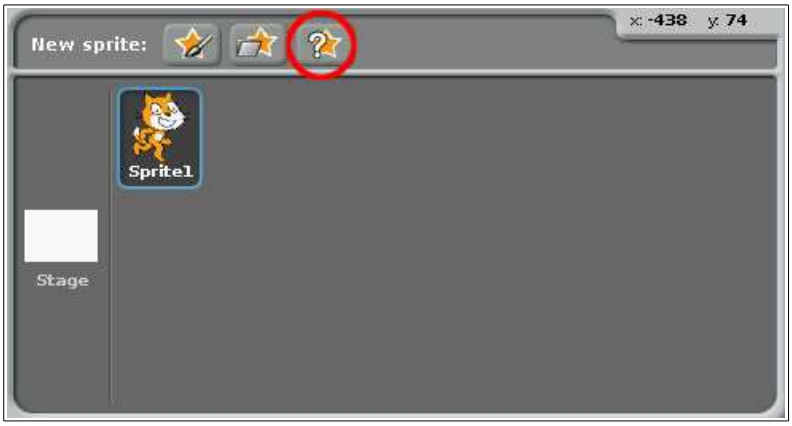
Click OK when done. This will drop your new sprite in the sprite list and on the stage:

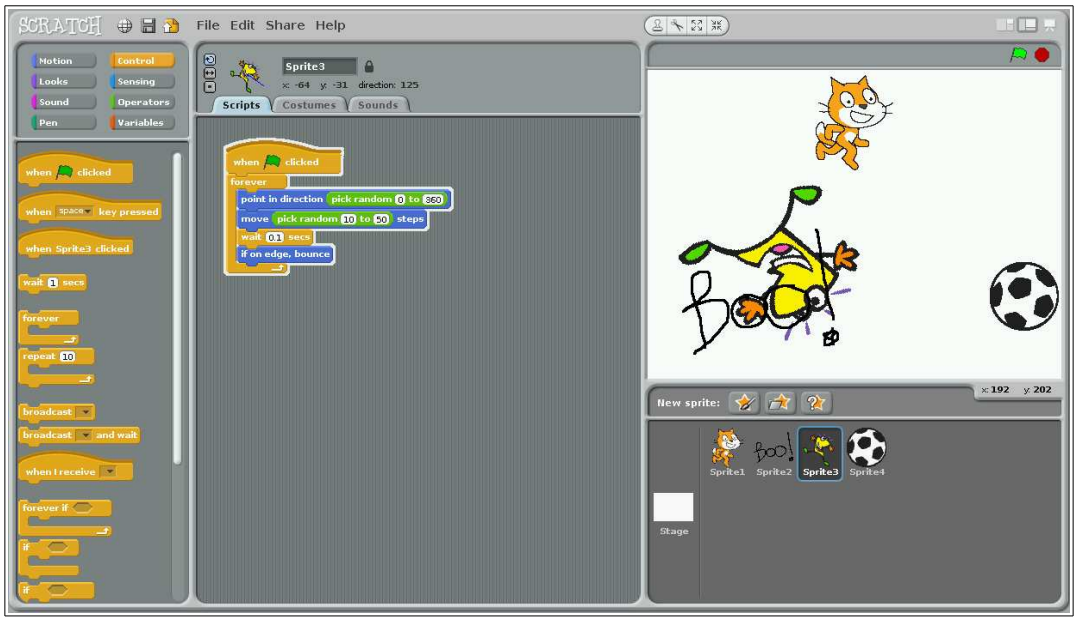
As mentioned earlier, you can also load a saved sprite if you click on the middle button in the row at the top of the sprite list:



Feel free to select any sprite from any category that you wish:

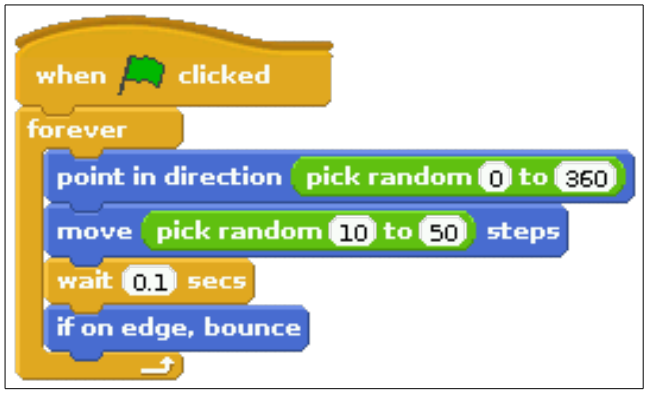
Again, you can have a random sprite brought to the stage too by clicking the right-most button:





You can click on any sprite in the sprite list, and its scripts load in the scripts area. This allows you to have a separate program for each sprite in the sprite list. Think about what this means. You can separately control each sprite while they all run their programs simultaneously!

Try clicking on one of the sprites in the sprites area and create the following program for it:

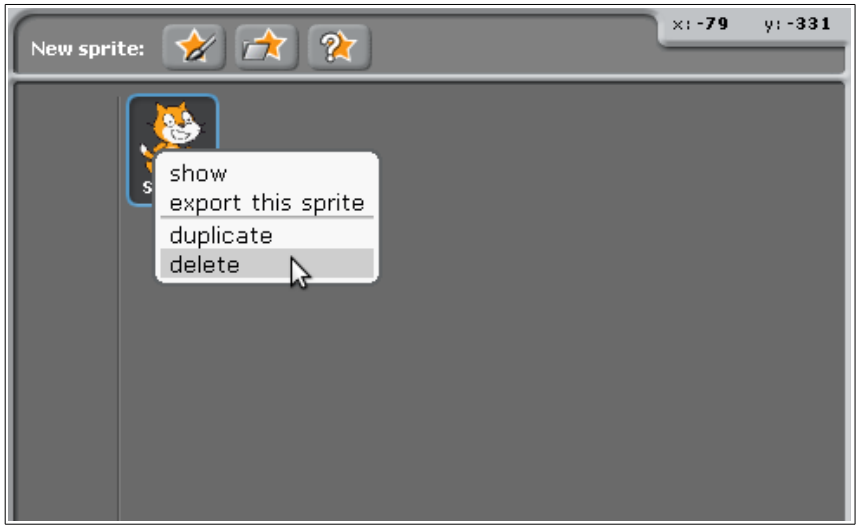


What does this program do? What does the **if on edge bounce** block do?

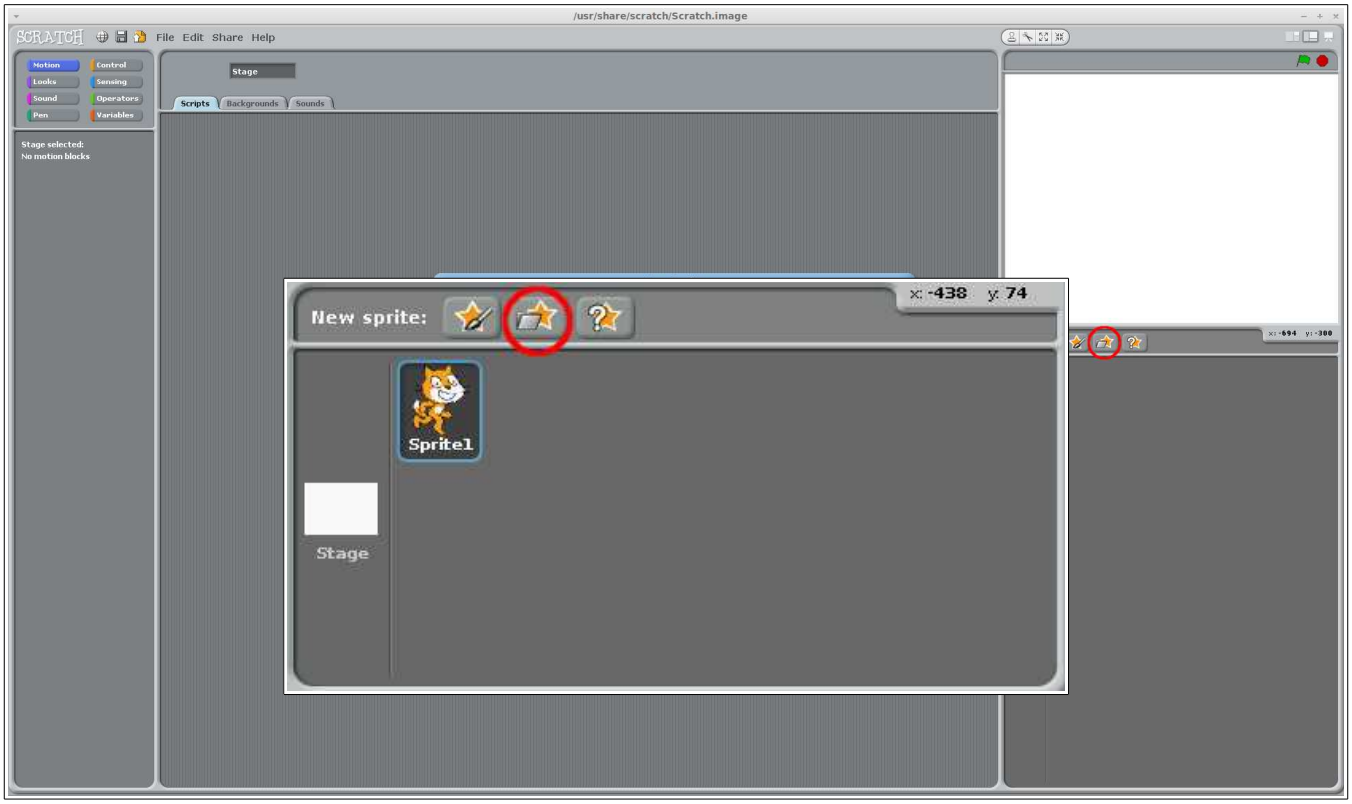
Catch the Baby!

The objective of the game *Catch the Baby!* is, well, to use a trampoline to catch a falling baby. Each time the baby is successfully caught by the trampoline, the player's score increases. The baby is randomly placed somewhere at the top of the screen, and then quickly descends to the bottom. The player can control the trampoline with the left and right arrow keys to position it below the falling baby. Technically, the baby can barely touch the trampoline to be saved. Here is a screenshot of the stage at the start of play:

The game
Start a new Scratch project. Remove the default cat sprite by right-clicking on the cat and selecting **delete**:



This is a good time to add the two sprites that you will need: the baby and the trampoline. To add the baby, click on the **choose new sprite from file icon**, then browse through the **People** folder and select the baby:



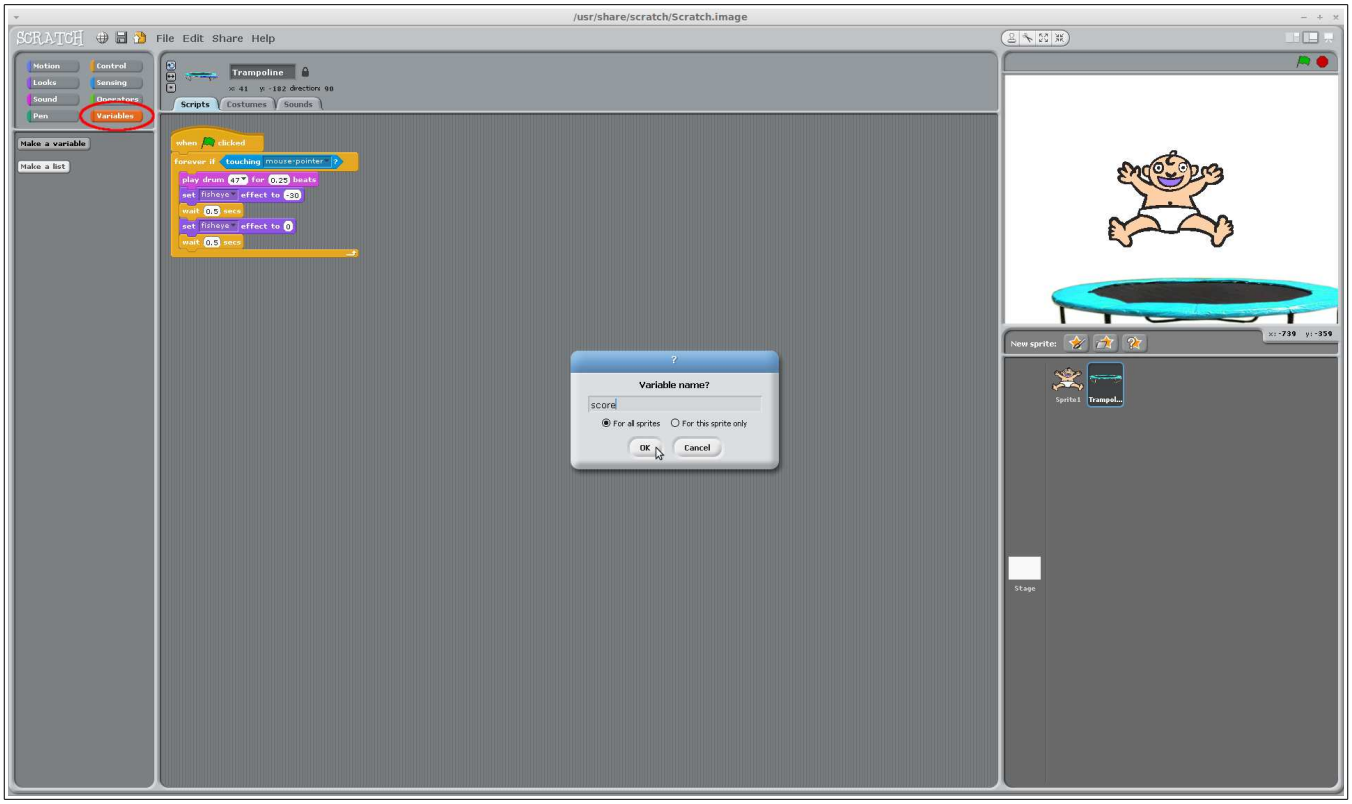
Rename the baby to something more appropriate, like **Baby**.

To add the trampoline, go through the same process, but browse through the **Things** folder and select the light blue trampoline:

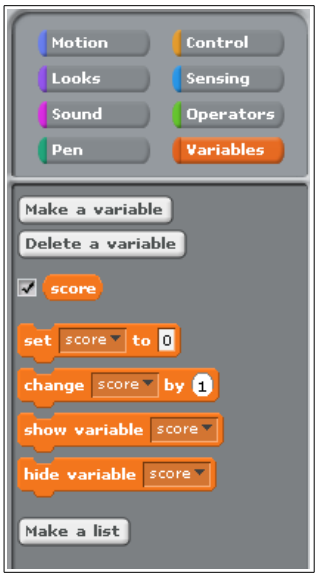
Note that the trampoline has a script associated with it. That is, it comes with a preloaded program that plays a short drum sound and changes the sprite if it is collide with. We will make use of some of this later. For now, resize the baby and trampoline sprites so that they are smaller (as in the image at the beginning of this activity).

Variables

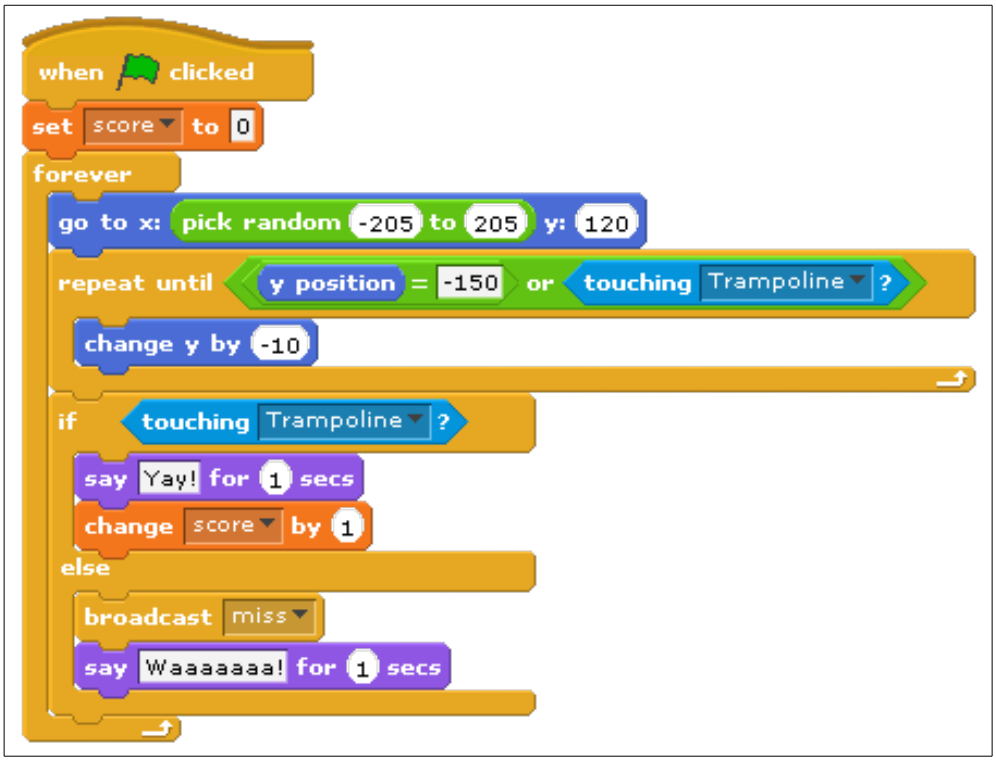
We will need one variable in our game: `score`. Add it now through the **variables** blocks group:



This adds the variable and allows us to modify it as we wish:



Make sure the baby sprite is selected in the sprites list and implement the following script for it:



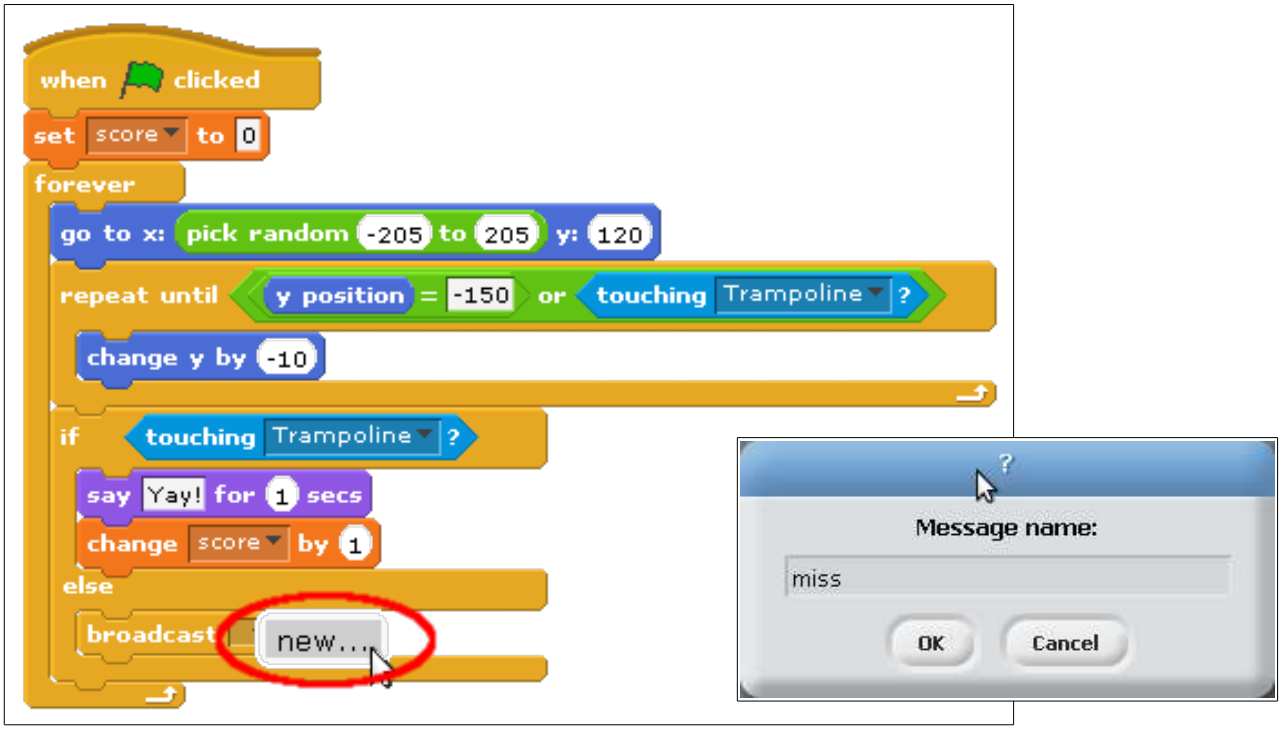
Let's explain what's going on here. This script runs when the green flag is clicked (i.e., when the game is started). The first statement sets the score to 0. Then, a group of statements is repeated forever (well, at least until the stop sign is clicked by the user).

The grouped statements in the **forever** construct first instruct the baby to move to a random position at the top of the screen (where y=120). By experimenting, it was calculated that the leftmost position for the baby should be at x=-205 and the rightmost at x=205.

At this point, a **repeat-until** construct is entered. Note that this is, in effect, repetition within repetition! The repeat-until condition instructs the baby to move down 10 pixels (**change y by -10**) until its is at position y=-150 or until it is touching the trampoline. In effect, it is instructing the baby to move down until it either collides with the trampoline or it reaches the bottom of the stage. Once either of these conditions occurs, the repeat-until construct is exited.

Note that we are using a literal value (y=-150) to detect when the baby reaches the bottom of the stage. This works fine so long as we initially place the baby at y=120 and repeatedly move the baby down 10 pixels at a time. That is, we can guarantee that the baby will eventually reach y=-150. But what would happen if the baby were initially placed at y=121 (or any value that isn't a multiple of 10)? Or if we decremented by 6 instead of 10 (i.e., **change y by -6**)? A better way may be to, instead, detect when the baby reaches a vertical position that is less than or equal to -150; that is, y<=-150.

The next statement is a selection statement in the form of an **if-else** construct. The script is now going to potentially do two different things, depending on whether or not the baby has collided with the trampoline. If it has (i.e., **touching Trampoline**), then it will say, "Yay!" for a bit, and the score will be incremented (since the baby was successfully caught by the trampoline). Otherwise (**else**), it will cry. Note the **broadcast miss** statement. This is a useful way to send another sprite a message. Any sprite can broadcast a message that other sprites can receive. In this case, the goal is to notify the trampoline that it has missed the baby. Adding a broadcast message is as simple as adding the block and creating a new message using the block's arrow:



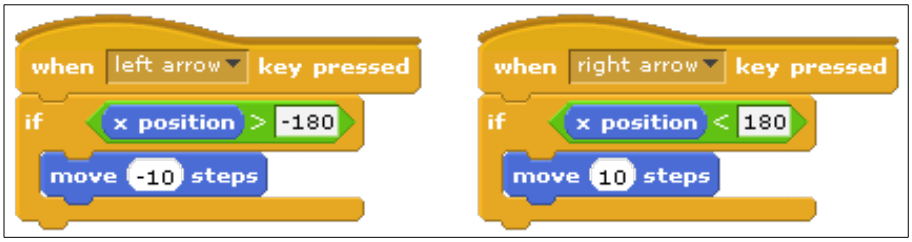
That's it for the baby! Now click on the trampoline in the sprites list and change the existing script to the following new script:

This change instructs the trampoline to move to the bottom-left corner of the stage when the green flag is clicked. It then repeats a set of statements forever, but only when it has collided with the baby (i.e., if **touching Baby**). If so, it first, it alters the trampoline sprite a little bit applying a *fisheye* filter (which makes the sprite appear to bend a little). A drum sound is then played, which is followed by a small delay, an undo of the *fisheye* filter, and another small delay.

Add another script to the trampoline as follows:

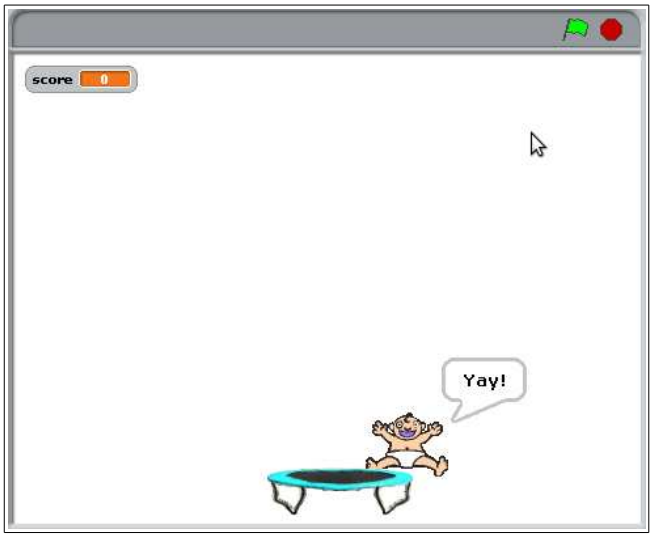
This script instructs the trampoline to say something appropriate when it receives the broadcasted message **miss**. Recall that this message was defined in the baby's script earlier. So the baby can broadcast the message which is then received by the trampoline. That is, if the baby reaches the bottom of the stage (i.e., the trampoline missed the baby), it broadcasts this message that the trampoline receives. This alerts the trampoline that it has missed the baby, and it utters an appropriate message.

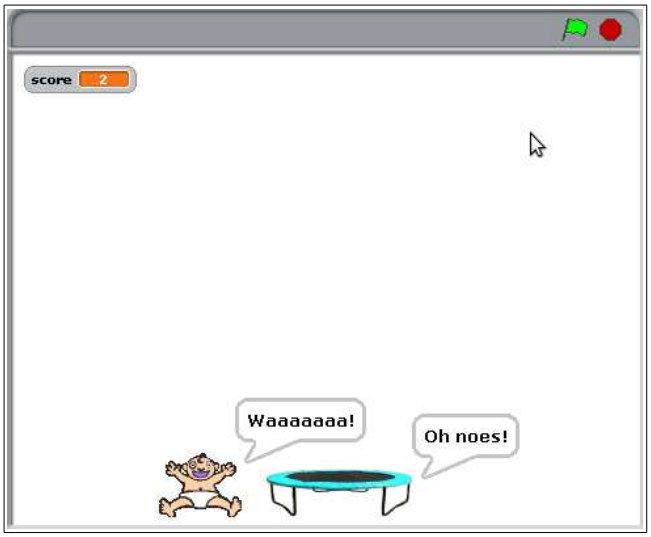
The last thing to add is the ability to move the trampoline with the left and right arrow keys. We can do this by adding the following two scripts:



These scripts instruct the trampoline to move 10 steps (to the left or right) when the arrow keys are pressed. In order to prevent the trampoline from going beyond the left or right border of the stage, however, additional **if** statements are added. These selection constructs prevent the trampoline from moving any further toward a border if it is already at one. For example, take a look at the left-arrow script. The left-most position for the trampoline is at x=-180. This was determined by testing (i.e., moving the trampoline with the mouse and capturing the x coordinate). The script checks to see if the trampoline's x-position is to the right of the established left-most position x=-180 (i.e., its **x-position > -180**). If so, it allows the sprite to move to the left; otherwise, it simply ignores the keypress.

At this point, you should be able to play the game by clicking on the green flag icon.





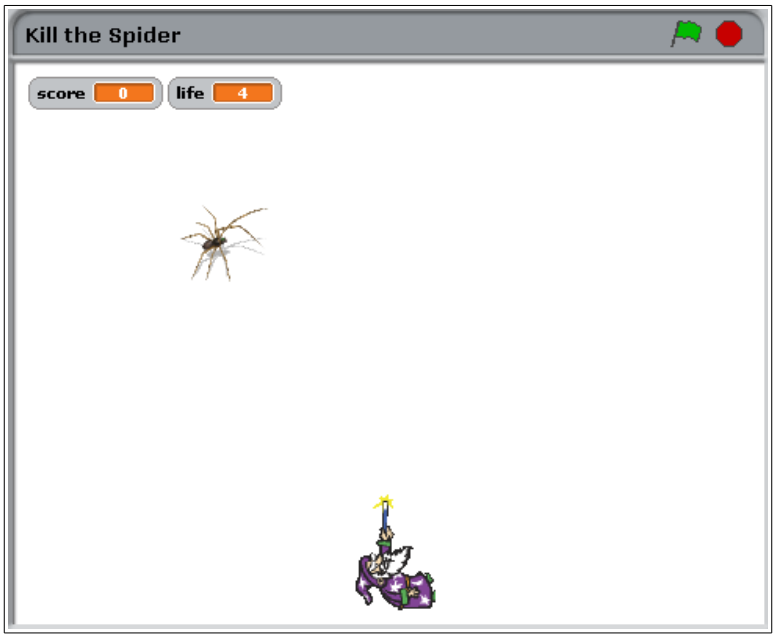
Click on the red stop sign icon to end the game.

Improvements
Although the game ends here, improvements can be made. For example, the baby could bounce in a random direction once it collides with the trampoline. Maybe it can defy the laws of gravity and move from side-to-side as it falls down. Experiment a bit.

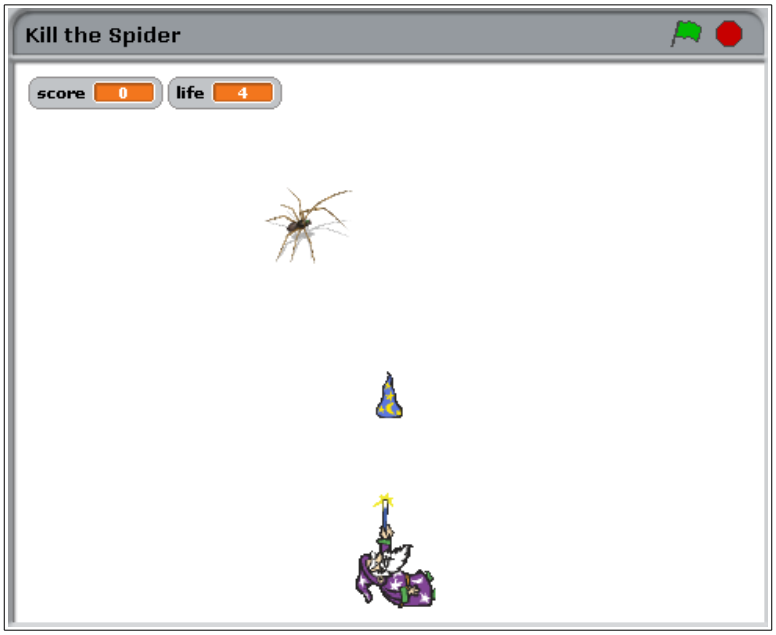
Homework: Kill the Spider!
In this activity, you will design a game based on *Catch the Baby!* Of course, it will be a bit different. **You are to submit your Scratch v1.4 (not v2.0!) file (with a .sb extension) through the upload facility on the web site.**

This game has two characters: a wizard and a spider. The wizard stands at the bottom of the stage and can move horizontally (just like the trampoline). The spider is placed to the left of the stage. At the beginning of the game, the wizard has five lives and a score of 0. Like the baby, the spider moves around the stage; however, it only does so horizontally. It is initially placed to the left of the stage at some random vertical position above the wizard. It then moves to the right until it reaches the right side of the stage.

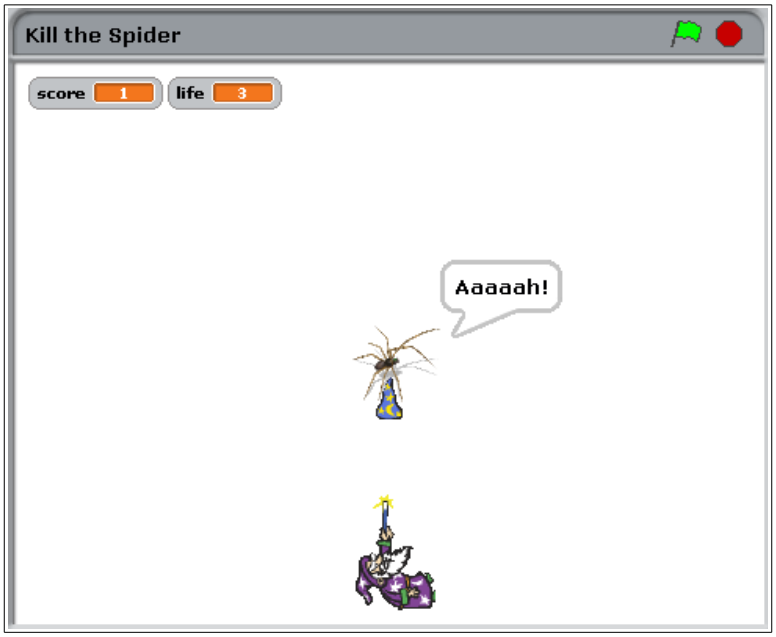
Spiders are scary and therefore should not be allowed to live. So the wizard can, of course, kill the spiders by shooting a wizard hat out of his wand. The player can make a hat shoot out of the wizard's wand by pressing the space bar. The hat starts at the wizard's wand and moves upward until it either collides with the spider or reaches the top of the stage. If the hat collides with the spider, the player's score increases by 1, and the spider reappears to the left of the stage at another random vertical position above the wizard for another round. If the spider is able to reach the right side of the stage, the wizard loses a life (since the spider was left to live and that is worthy of losing a life). The game ends when the wizard uses up all of his lives. Here's what the stage looks like during the game:



Here's what it looks like when the wizard shoots a hat:



And here's what it looks like when the wizard kills a spider:



Code your game in Scratch according to the description above. Feel free to add extra features or embellishments (although this is not a requirement) for **up to five bonus points**. At minimum, your game should feature a wizard that can shoot a hat from his wand in order to kill spiders. The wizard should move via the left and right arrow keys, and should shoot a hat from his wand via the space bar. If the hat collides with a spider, the spider should say something appropriate, another spider should reappear to the left of the stage, and the game should continue. Each time the wizard kills a spider, the score should increase by one. Each time a spider reaches the right of the stage, the wizard should lose one life. Start your game with five lives.

A note about the spider sprites: You can use your own spider image (e.g., a JPG or PNG). You can find some on the Internet through Google image search. Feel free to edit the image as you see fit. In Scratch, you can access the image and add it as a sprite the same way that you normally do to add sprites already in Scratch (i.e., via the **choose new sprite from file** icon). You will need to browse to the location where your spider image was saved. The spider sprite used in the example above is actually made up of 16 individual images so that it can appear to be crawling as it moves across the stage. The first image was selected as the sprite. The other 16 were added as separate costumes by selecting the costumes tab above the scripts area.

Raspberry Pi Activity: LED the Way

In this activity, you will implement various circuits, primarily using LEDs, resistors, and push-button switches. The Raspberry Pi will initially be used solely as a power supply; however, as the activity progresses, you will use it to programmatically affect the circuits you create. You will need the following items:

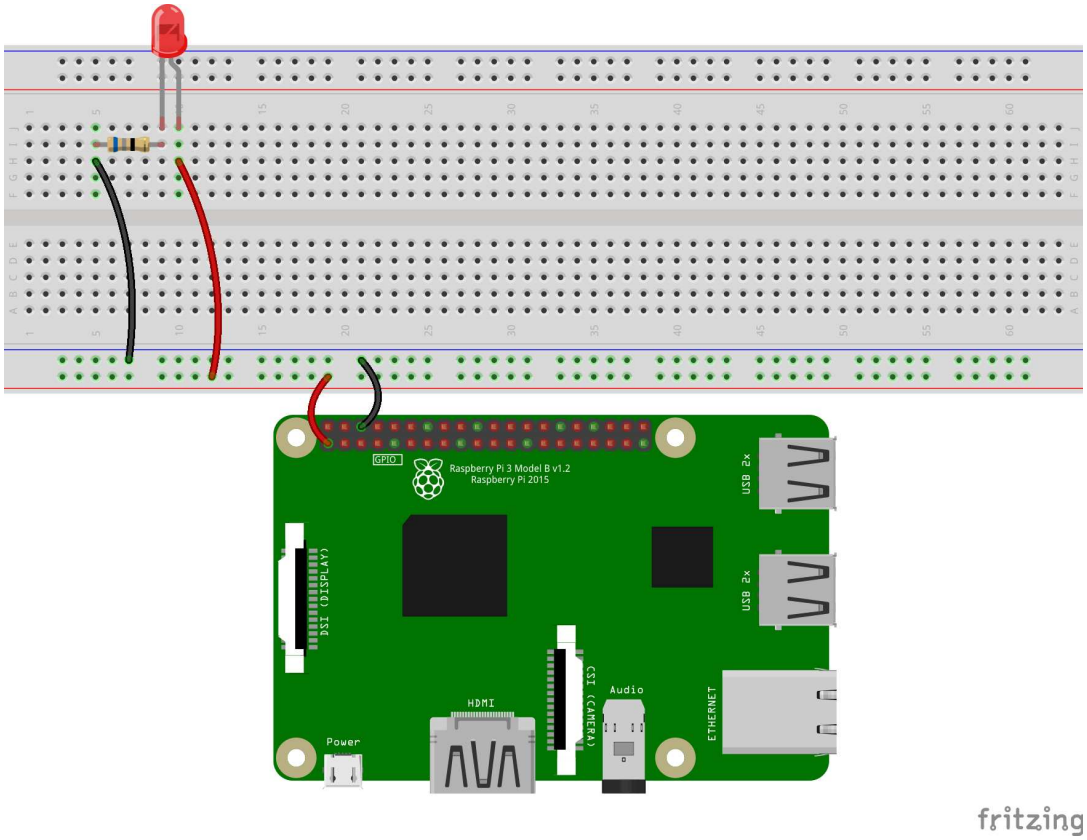
- Raspberry Pi B v3 with power adapter;
- LCD touchscreen;
- Keyboard and mouse;
- Breadboard;
- GPIO-to-breadboard interface board with ribbon cable; and
- LEDs, resistors, switches, and jumper wires provided in your kit.

GPIO

This is the first activity in which the GPIO (General Purpose Input and Output) pins will be used. Recall that these pins allow various inputs to and outputs from the RPi to be utilized in external circuits (typically implemented on a breadboard).

A simple circuit

The first circuit you will implement is the very simple one shown in an earlier lesson (the topic of which was computer architecture). Here was the layout diagram of that circuit:



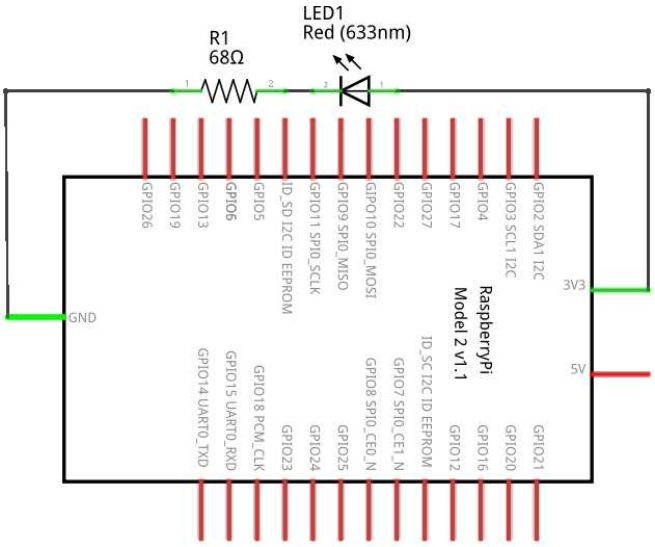
In the diagram, the red wire is connected on one end to a GPIO pin that exposes a 3.3V (3V3) power source. The other end is connected to the bottom row of the breadboard. Recall that this row is internally connected horizontally. Therefore, all holes in that row now have 3.3V.

The black wire is connected on one end to a GPIO pin that exposes ground (GND) or 0V. The other end is connected to the top row in the bottom section of the breadboard. Ground is therefore provided to all holes in that row.

The red LED is inserted so that its legs span across several columns in the center of the breadboard. Recall that these columns are internally connected vertically (however, there is a disconnect across the center gap). Note that the columns are numbered. So for example, the holes in column 20 are internally connected on either side of the center gap (but not to each other across the gap).

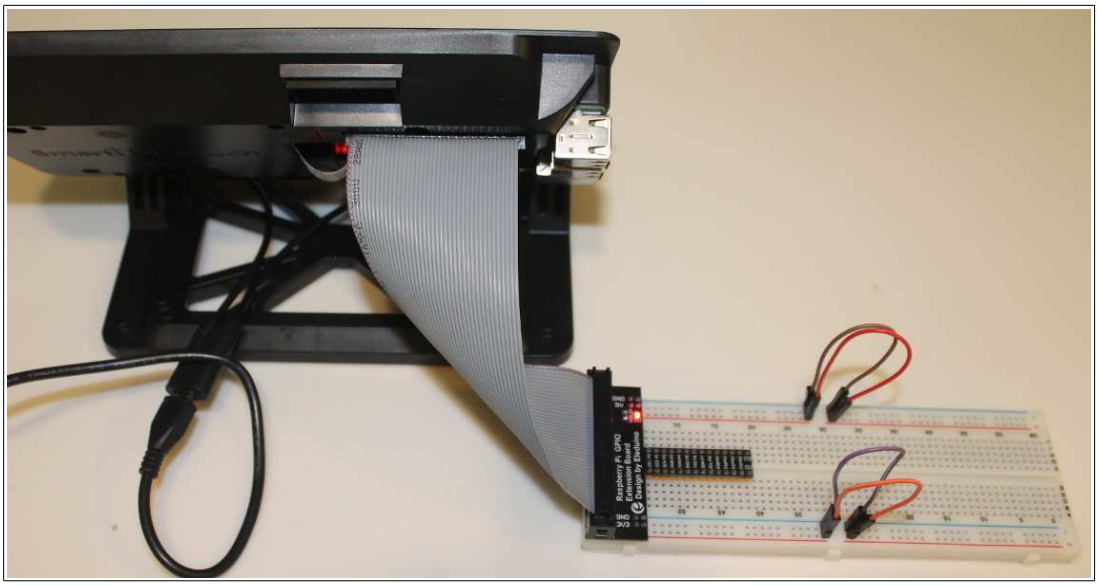
A red wire connected the 3.3V power source from the bottom row to the column that has the positive side of the LED (the long leg – or anode). A 68Ω resistor then connects the negative side of the LED (the short leg – or cathode) to ground. It does so by being placed across several columns (from the negative side of the LED to another column), and then by use of a black wire that brings ground to the other side of the LED.

In all, the circuit requires a red LED, a 68Ω resistor, and some jumper wires (oh, plus the RPi). Here is the circuit diagram:

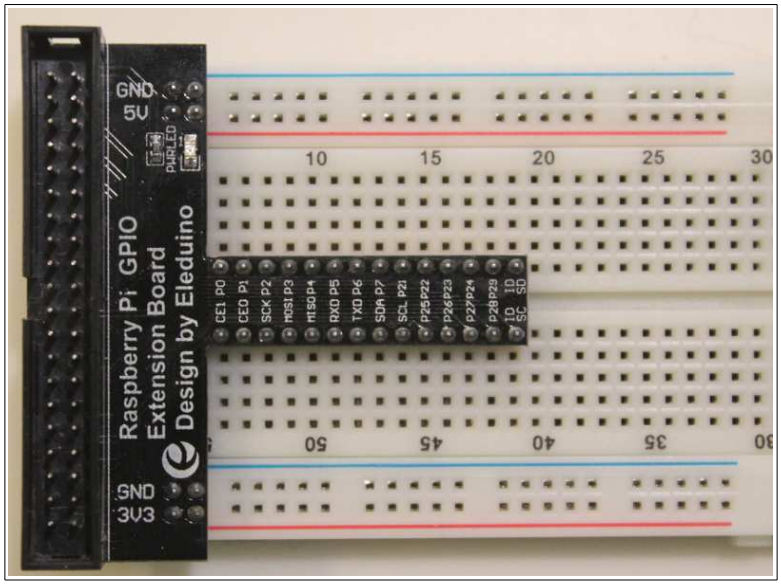


The 68Ω resistor has the colored bands: blue, gray, black. **Note that your kit does not come with 68Ω resistors!** The closest one in your kit is a 220Ω resistor (which will work just fine). It has the colored bands: red, red, brown. For the circuits in this activity, you will use a 220Ω resistor.

Since the power source for the circuit will come from the RPi, we need a way to connect the GPIO pins to the breadboard. One way, as in the layout diagram above, is to connect wires to the GPIO pins and the breadboard. The problem is that the wires in your kit aren't particularly well suited for this. Your kit does, however, include a GPIO interface board that can extend the GPIO pins to the breadboard using a ribbon cable:



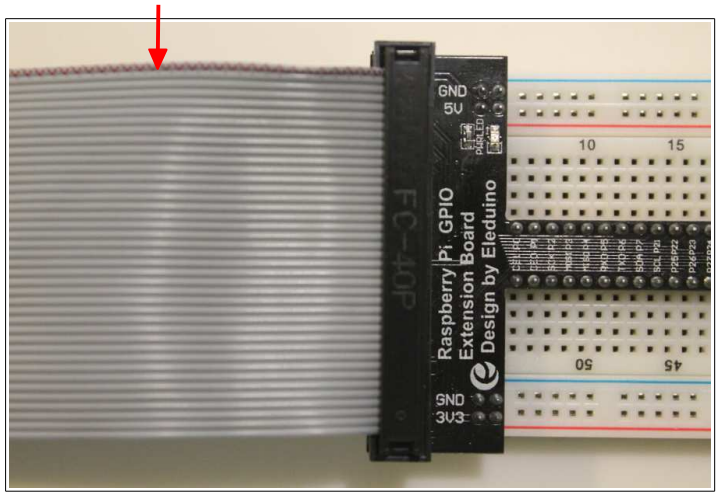
The GPIO interface board extends the GPIO pins to the central holes on the breadboard. First, place the GPIO interface on the breadboard as shown below:



Note that your kit may come with a different GPIO interface board. Perhaps it's a different color (e.g., green vs. black) or has a different pin layout. This activity includes directions for the various GPIO interface boards that may be part of your kit.

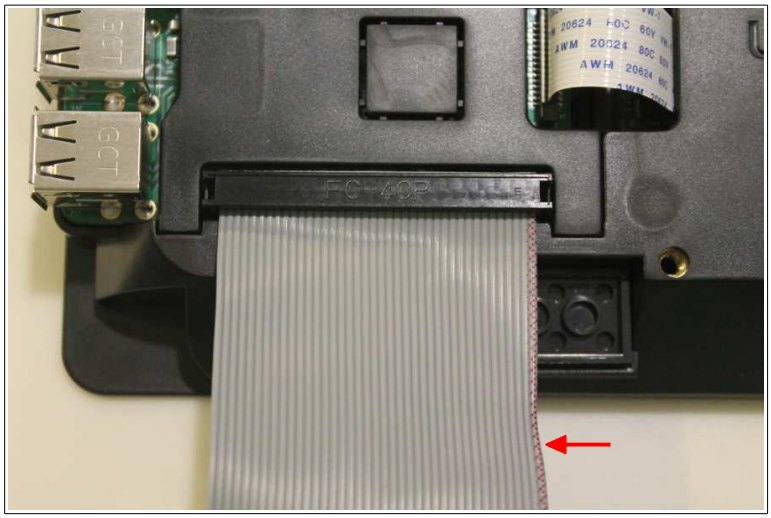
Note the orientation of the breadboard: the positive rails are on the bottom of each section at the top and bottom of the breadboard. Also, the entire GPIO interface rests on the breadboard (i.e., the left part of the GPIO interface in the image above does not hang over the edge of the breadboard), and the central pins straddle the gap in the center of the breadboard. When pressing the interface board into the breadboard, make sure to put even downward pressure entirely across it to prevent it from breaking.

Next, connect the ribbon cable to the GPIO interface as show below:

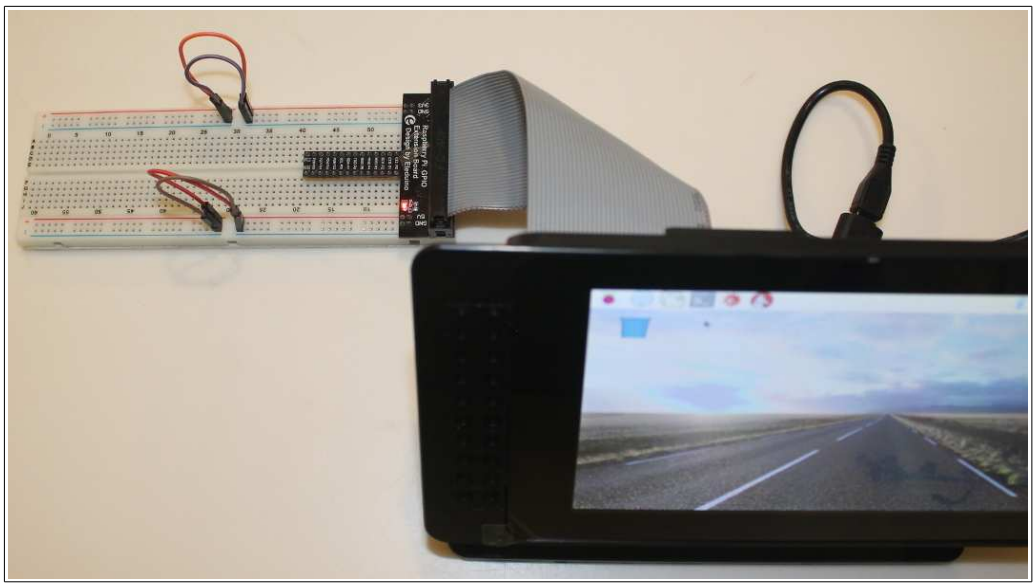


Notice how the **red edge** of the ribbon cable (as noted by the arrows) is aligned with the top of the breadboard. There's also a tab on the hard plastic end of the ribbon cable that prevents it from being inserted incorrectly into the GPIO interface.

Next, connect the other end of the ribbon cable to the GPIO pins on the RPi that are exposed at the rear of the stand:



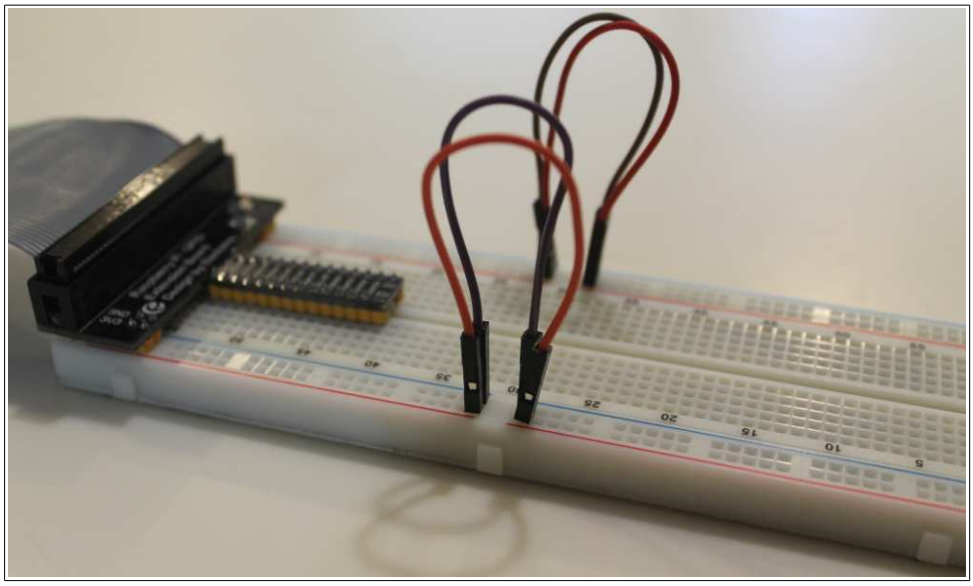
Again, notice the orientation of the ribbon cable! The best way to lay everything out is shown below:



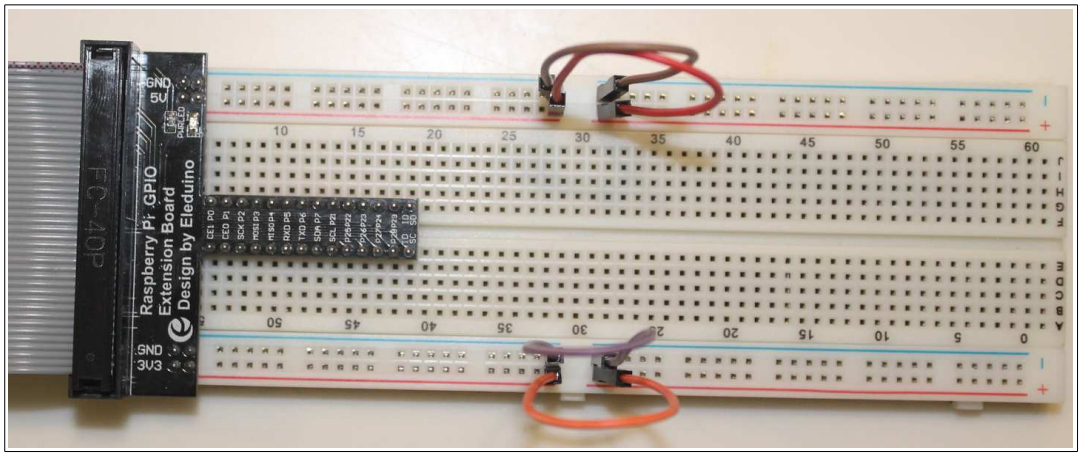
If you've connected everything correctly, a little red LED on the GPIO interface should be on.

The GPIO interface allows circuits to be connected to the RPi's GPIO pins. It also exposes +5V, +3.3V, and GND. Viewed as before (where the GPIO interface is to the left of the breadboard), 5V and GND are on the top rails, and 3.3V and GND are on the bottom rails.

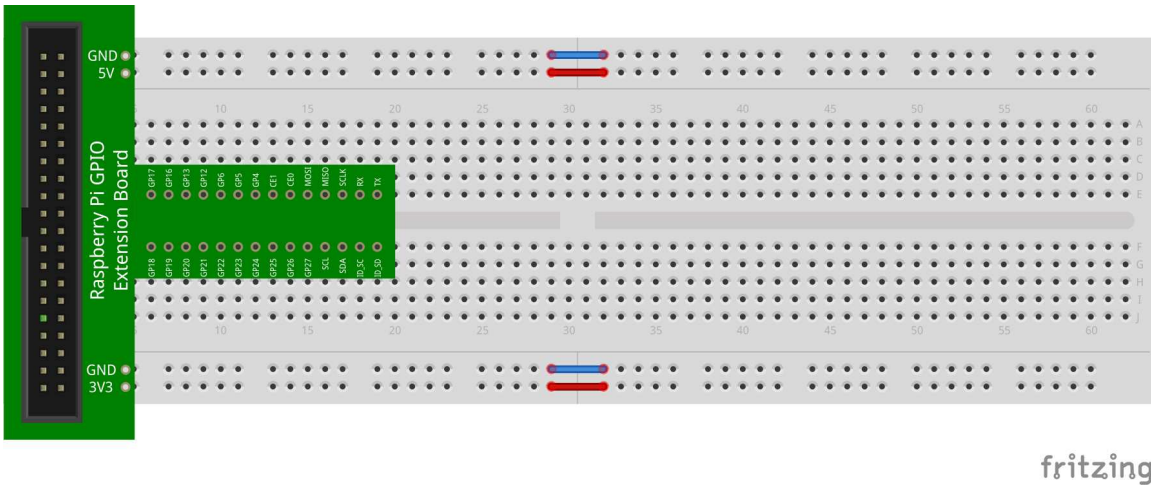
A word of caution, however! Your breadboard may not be internally connected across the entire top and bottom rails. If not, you will need to bridge the left and right halves of the rails as shown below to ensure that power and GND are exposed across the entire length of the rails:



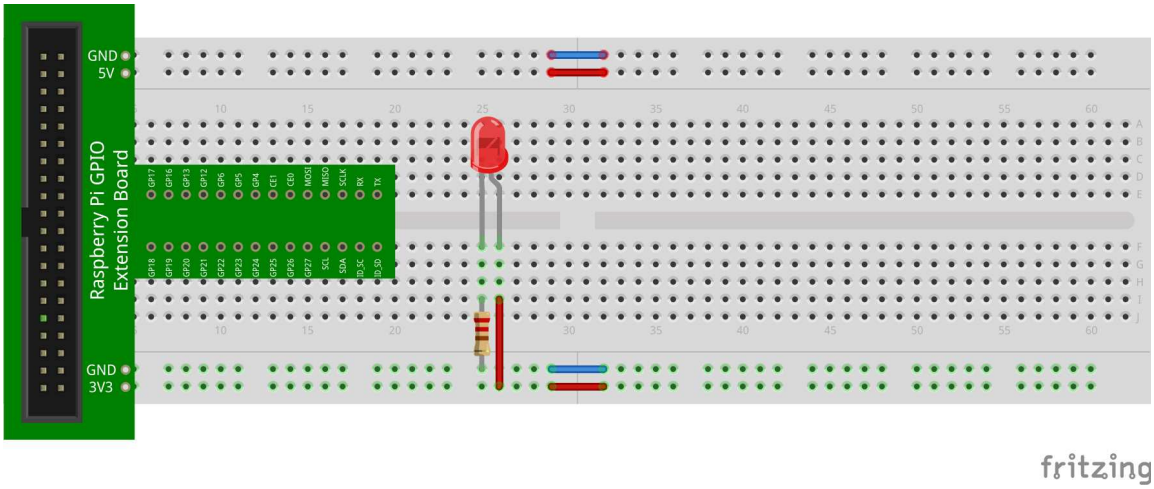
This must be done for both 5V and GND in the top rails, and 3.3V and GND in the bottom rails:



Here's a layout digram of this:



We can now create a layout diagram of the circuit shown earlier a bit to make use of the interface board:



Note that it is assumed that a ribbon cable connects the interface board to the GPIO pins on the RPi (i.e., it is not shown in the layout diagrams).

Create the circuit shown in the layout diagram above **without connecting the power adapter to the RPi yet**. Make sure that:

- The LED straddles two columns (i.e., does not have both of its legs in the same column of holes);
- You connect a wire from a 3.3V pin on the bottom rail to the positive side (long leg) of the LED; and
- You place a 220Ω resistor from the negative side (short leg) of the LED to ground (note that the GPIO interface board brings ground to **both top rows** of the top and bottom rails).

To summarize, use a red (or similar) wire to connect a 3.3V power source from the interface board to the positive side of the LED. Use a 220Ω resistor to connected the negative side of the LED to GND. When you are certain that your circuit is correct, plug in the RPi. If everything is wired correctly, the LED should light.

Calculations

The resistor used in the circuit is a 220Ω resistor, the source voltage is 3.3V, and the LED has a forward voltage (i.e., voltage drop) of 2V (which we know from reading its data sheet). Using Ohm's Law, we can calculate the current that will flow through the circuit as follows:

$$\begin{aligned} V &= I * R \\ (3.3V - 2V) &= I * 220\,\Omega \\ 1.3V &= I * 220\,\Omega \\ 0.00591\,A &= I = 5.91\,mA \end{aligned}$$

The current through the LED will be 5.91mA. This is much less than the recommended 20mA. Note that the brightness of the LED is directly related to how much current flows through it. The more current, the brighter it will be. Of course, there is a limit (as per the data sheet).

We can calculate the power dissipated by the resistor as follows:

$$\begin{aligned} P &= V * I \\ P &= (3.3V - 2V) * 0.00591\,A \\ P &= 1.3V * 0.00591\,A \\ P &= 0.0077W = 7.7\,mW \end{aligned}$$

The resistor in your kit is a 1/4W (250mW) resistor. It is more than enough. The power dissipated by the LED can be calculated similarly:

$$\begin{aligned} P &= V * I \\ P &= 2V * 0.00591\,A \\ P &= 0.0118\,W = 11.8\,mW \end{aligned}$$

The LED in your kit has a forward current limit of 120mW. Again, it is more than enough.

Increasing the voltage

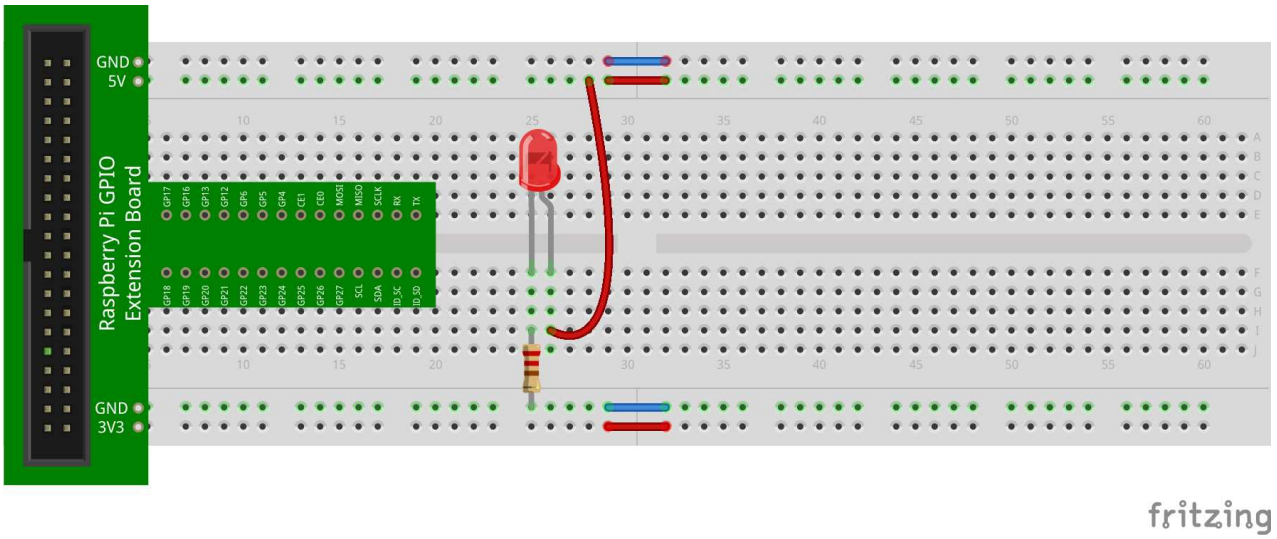
To make the LED a bit brighter, we cannot reduce the resistance in the circuit since there aren't any lesser-valued resistors in the kit. We can, however, change the source voltage! The RPi also has a 5V power source. Suppose you were to, instead, use the 5V power source from the RPi. This would provide more voltage to the breadboard and across the circuit. According to Ohm's Law, if we increase the voltage and keep the resistance in the circuit at 220Ω, the current has to increase. In the space below, calculate the current that would flow through the circuit with a 5V power source and a 220Ω resistor:

Now, calculate the power dissipated by the resistor for the 5V power source:

And finally, calculate the power dissipated by the LED for the 5V power source:

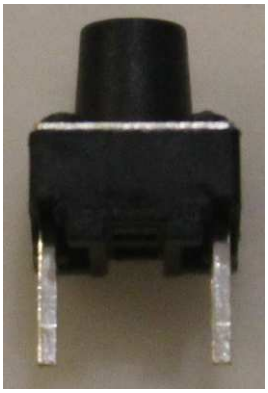
So with the 220Ω resistor and a 5V power source, we increase the current – which should make the LED appear a bit brighter when lit.

Alter your circuit as in the figure below. The only difference is that the positive side of the LED should now be connected to 5V instead of 3.3V:



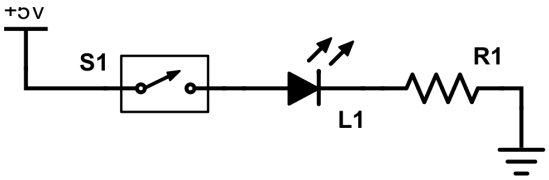
Adding a push-button switch

Let's add a push-button switch to the above circuit. The switch will control the flow of electricity to the circuit. If the button is pushed, current will flow and the LED will light. A push-button switch is a *tactile* switch that usually has two to four legs. The switch in your kit has two legs:

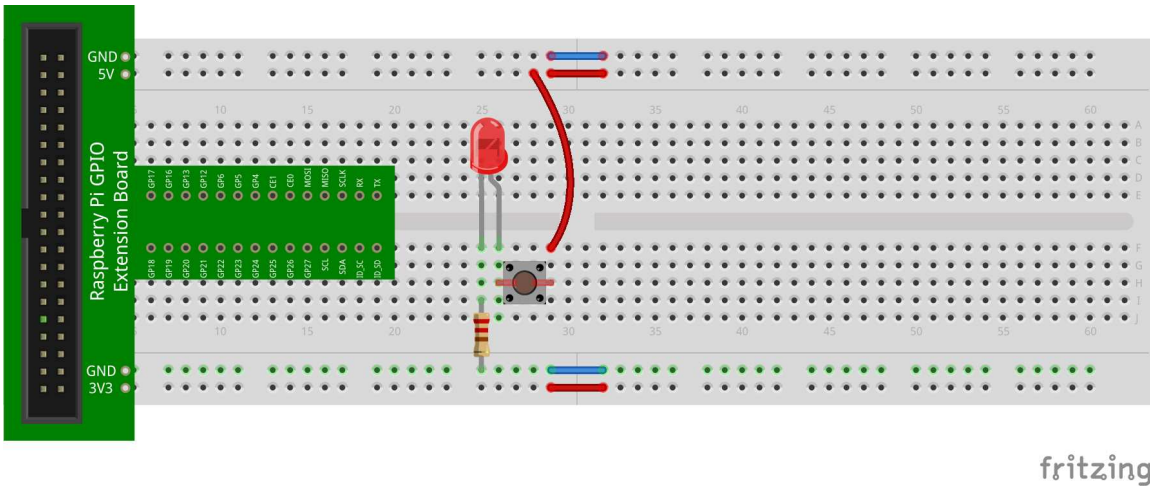


This type of switch is typically positioned across several columns in the central part of the breadboard. That is, the pins should be in separate columns. Power is connected to one pin. The part of the circuit to be powered when the switch is pushed is connected to the other pin.

Modify your circuit by adding a switch as indicated below:



Here's a layout of this circuit:



Note that one pin of the switch is connected to +5V, and the other to the positive side (long lead) of the LED.

An interesting experiment is to see the difference in LED brightness when connected to 5V vs. 3.3V. We can quickly calculate the amount of current flowing through the LED in both cases, with a constant resistance of 220 Ohms (also assuming that the voltage drop across the LED is 2V). First, with 5V:

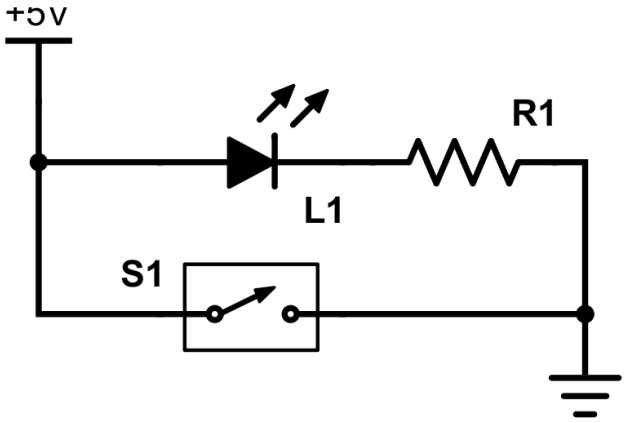
$$
\begin{aligned}
V &= IR \\
5V - 2V &= I(220\ \text{Ohms}) \\
3V &= I(220\ \text{Ohms}) \\
I &= 3V / 220\ \text{Ohms} \\
I &= 0.014\ A = 14\ mA
\end{aligned}
$$

Now, with 3.3V:

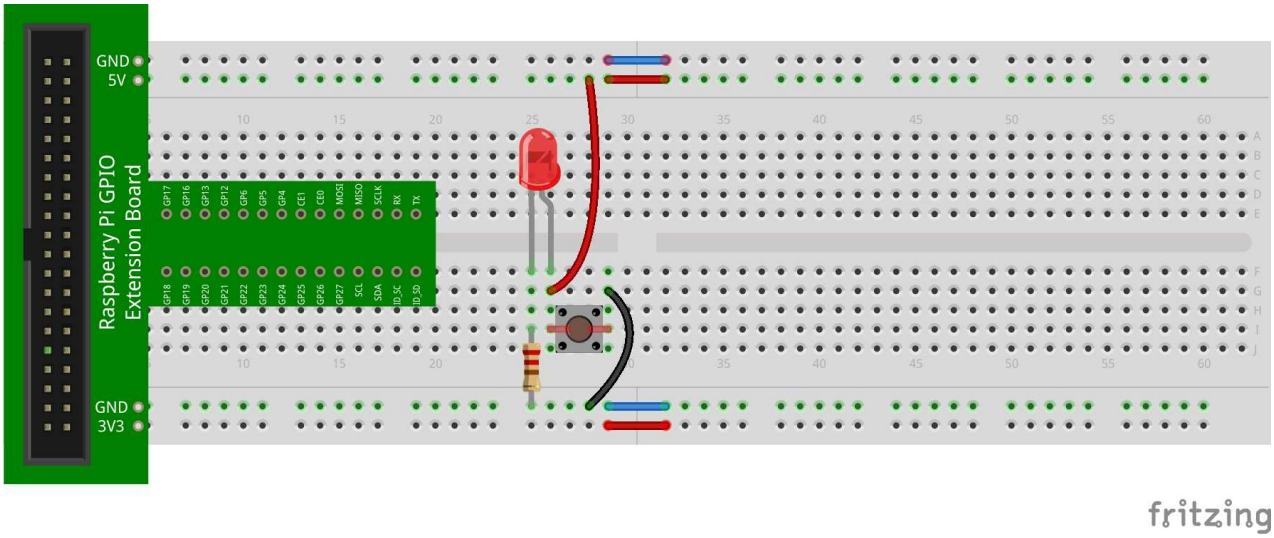
$$
\begin{aligned}
V &= IR \\
3.3V - 2V &= I(220\ \text{Ohms}) \\
1.3V &= I(220\ \text{Ohms}) \\
I &= 1.3V / 220\ \text{Ohms} \\
I &= 0.006\ A = 6\ mA
\end{aligned}
$$

The LED's brightness should be noticeably different!

Another interesting thing to do is to watch how electricity takes the path of least resistance. Let's provide two paths for +5V to flow: (1) through the LED, to a resistor, and finally to ground; and (2) through a switch to ground. Here's the circuit:



Here's one way to layout this circuit:



When the switch is open, current flows through the LED and resistor. However, when the switch is closed, current flows through the switch and directly to ground. Since this is the path of least resistance (i.e., there are no components to resist current – other than the switch), the LED turns off. That is, no current flows through the LED and resistor. **Be careful!** There is a lot of current flowing through the switch; therefore, a lot of power is dissipated (as heat). **Make sure to press the switch for just a moment so that you don't damage it.** In fact, we can calculate the current and power in the circuit:

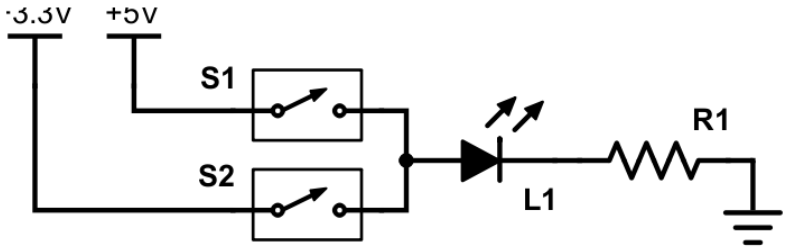
$$V = IR$$
$$5V = I(0.01\ \text{Ohms})$$
$$I = 5V / 0.01\ \text{Ohms}$$
$$I = 500A!$$

Does it seem possible that 500A is flowing through the circuit? In fact, no. The power supply included with your kit produces, at most, approximately 2.5A. If we assume that 2.5A is flowing through the switch portion of the circuit, we can calculate the power dissipated by the switch:

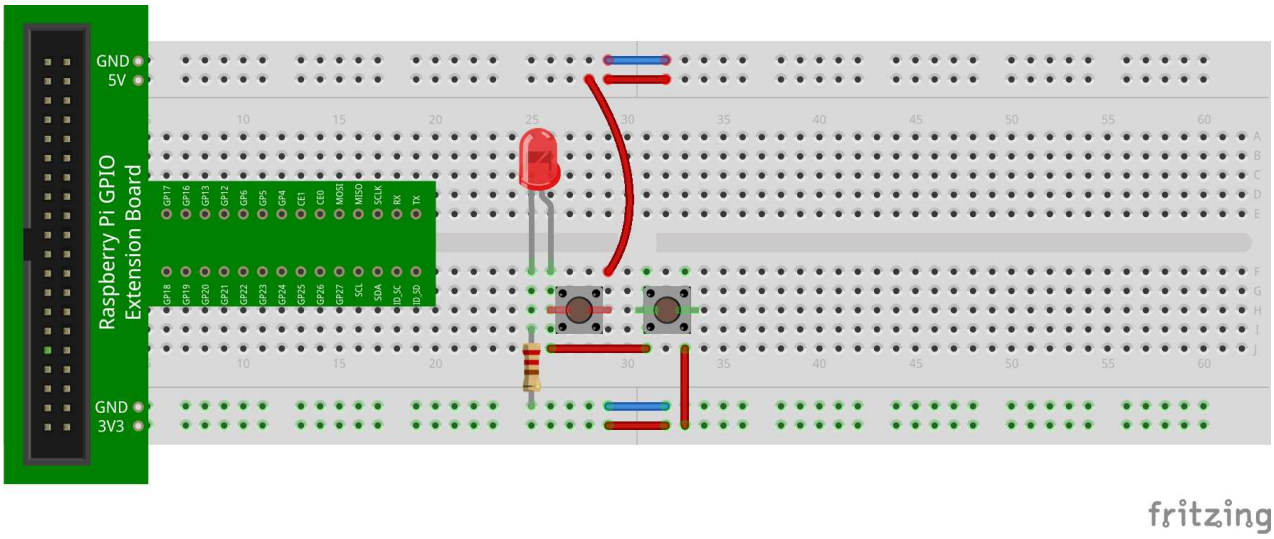
$$P = VI$$
$$P = 5V(2.5A)$$
$$P = 12.5\ W$$

The power dissipated by the switch is 12.5W. Most pushbutton switches have a maximum rating of 50mA at 12V (or 0.6W). You can see why pressing the switch for more than a moment could significantly damage it!

Earlier you noticed how the LED brightness was different when it was supplied with 3.3V vs. 5V. Let's try out another experiment to further illustrate this. Implement the following circuit:



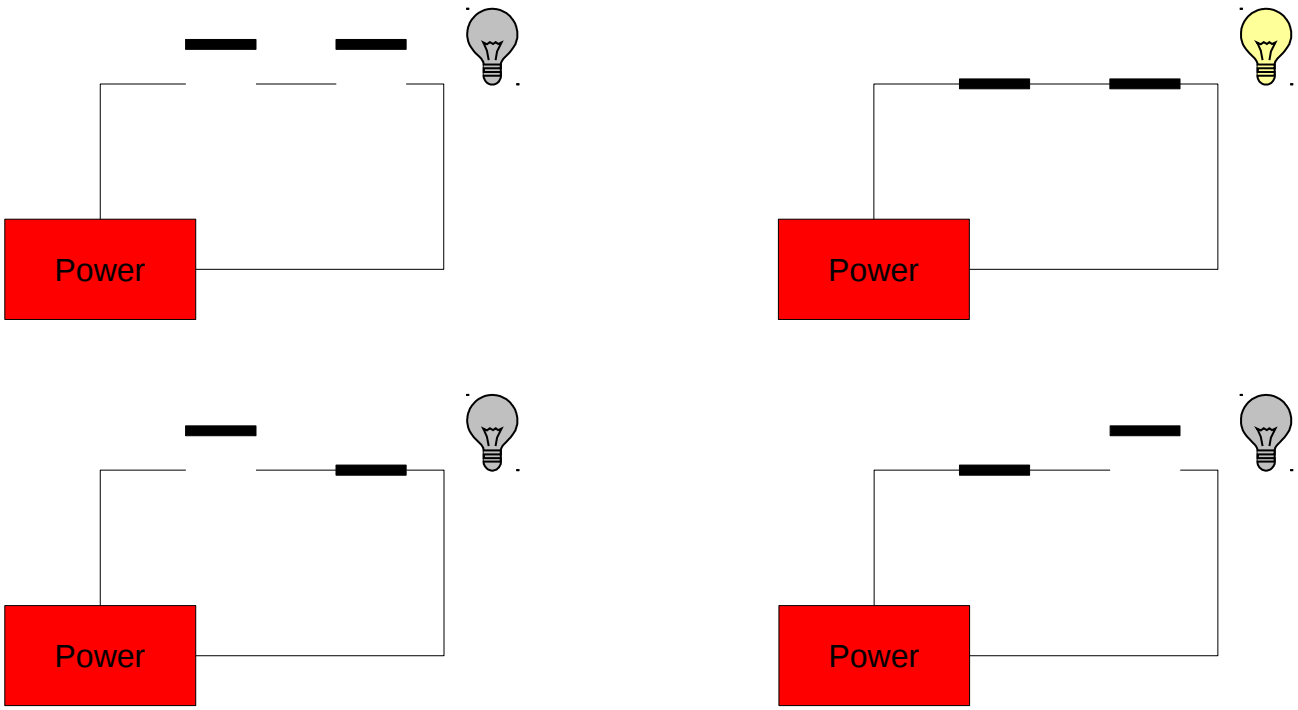
Here's one way to layout this circuit:



To test, experiment by pressing the switches in an alternating fashion.

Recall that the circuits in a previous lesson also included versions with multiple switches (both in parallel and in series). We later related this to logic gates. The rest of this activity will have you experiment with various configurations of multi-switch circuits.

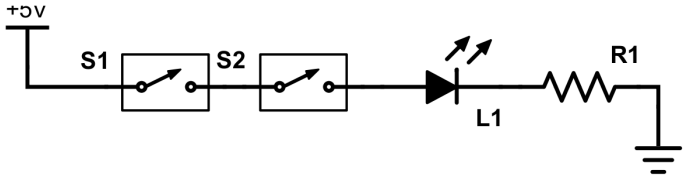
Replicating the *and* gate
Recall the following circuit:



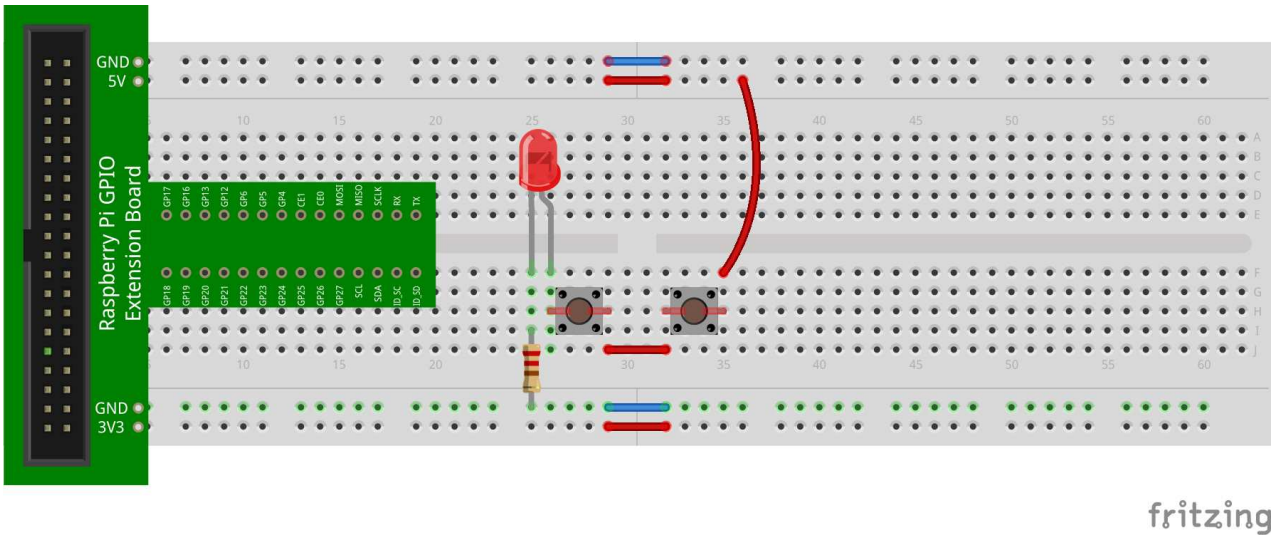
This circuit has two switches in *series*. Placing switches in this configuration in the circuit replicates the functionality of the *and* gate. Fill in the truth table for the *and* gate below:

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

The output, *Z*, is only 1 (true) when both inputs, *A* and *B*, are 1 (true). In the circuit above, the light bulb is on (1) when both switches are closed (1). To implement this in your LED circuit, modify it as follows:



Here's one way to layout the circuit:

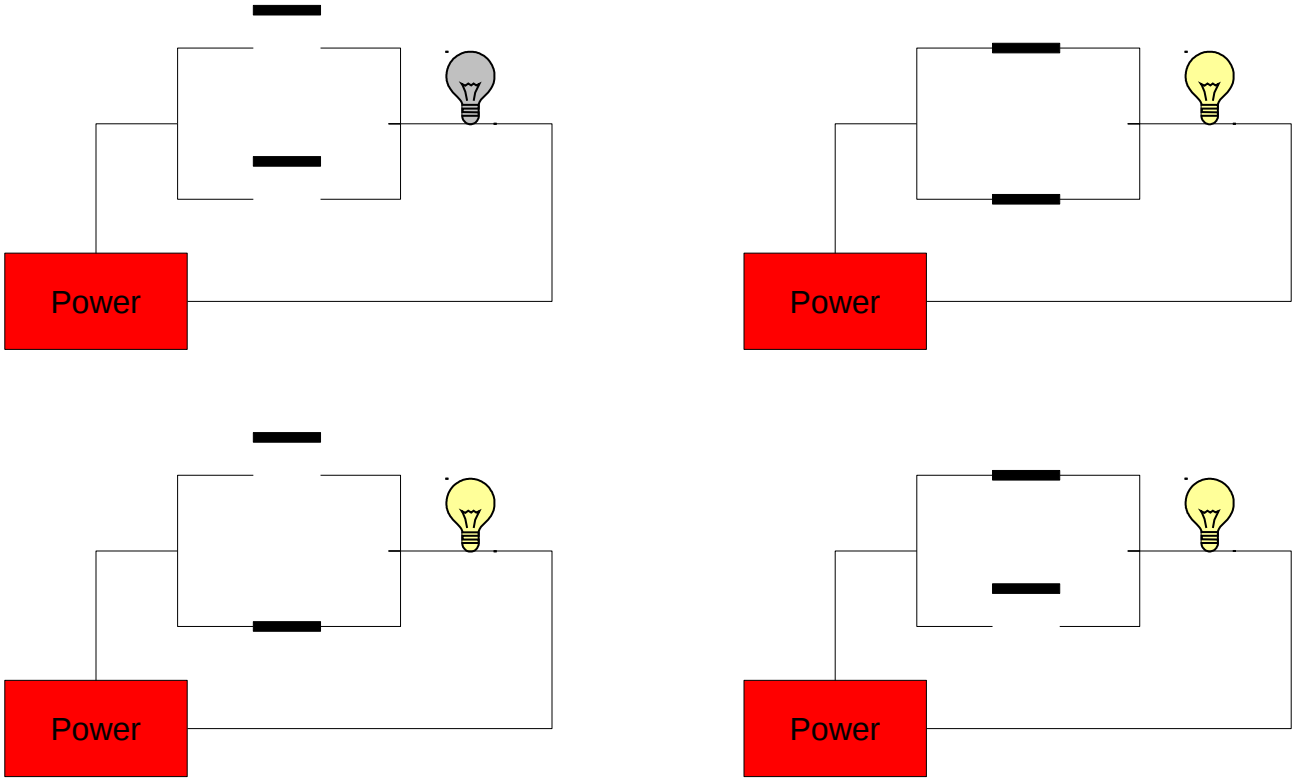


Power is extended to one of the switches (via the long red wire). This switch is then connected to the second switch. That is, this switch's left pin (in the figure above) is connected to the second switch's right pin. The second switch's left pin is connected to the positive side of the LED (they are in the same column). The rest of the circuit (resistor from the negative side of the LED to ground) is the same.

Try the circuit. The LED should only light when *both* switches are closed.

Replicating the *or* gate

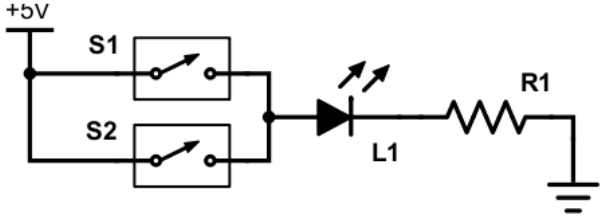
We can also implement the functionality of the *or* gate as follows:



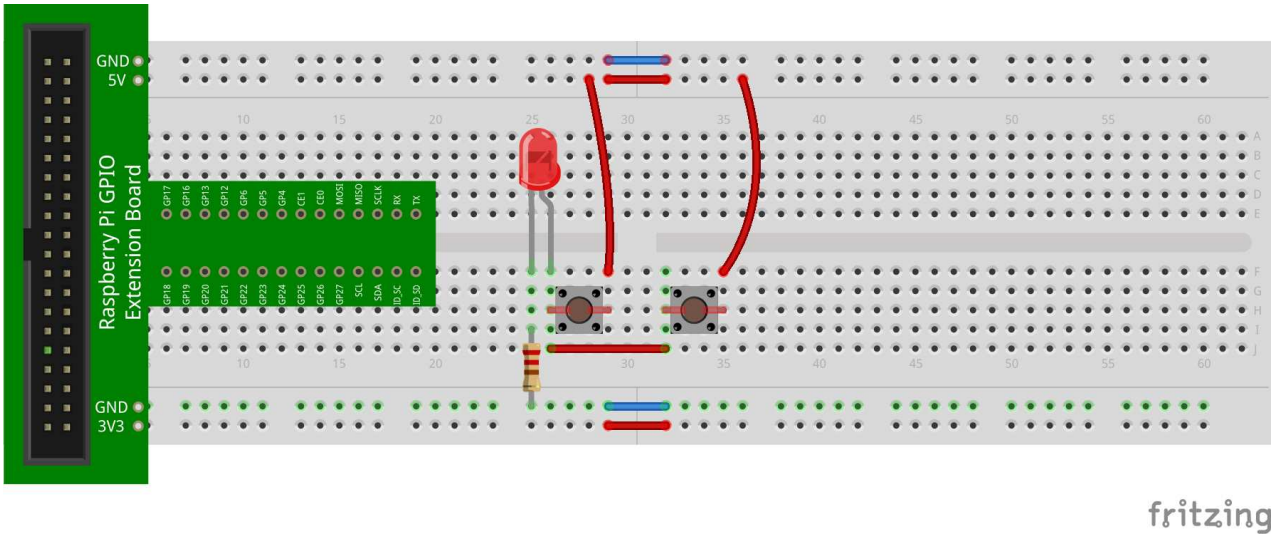
This circuit has two switches in *parallel*. Placing switches in this configuration in the circuit replicates the functionality of the *or* gate. Fill in the truth table for the *or* gate below:

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

The output, *Z*, is 1 (true) when any input (*A*, *B*, or both *A* and *B*) are 1 (true). In the circuit above, the light bulb is on (1) when either switch (or both) are closed (1). To implement this in your LED circuit, modify it as follows:



Here's one way to layout the circuit:



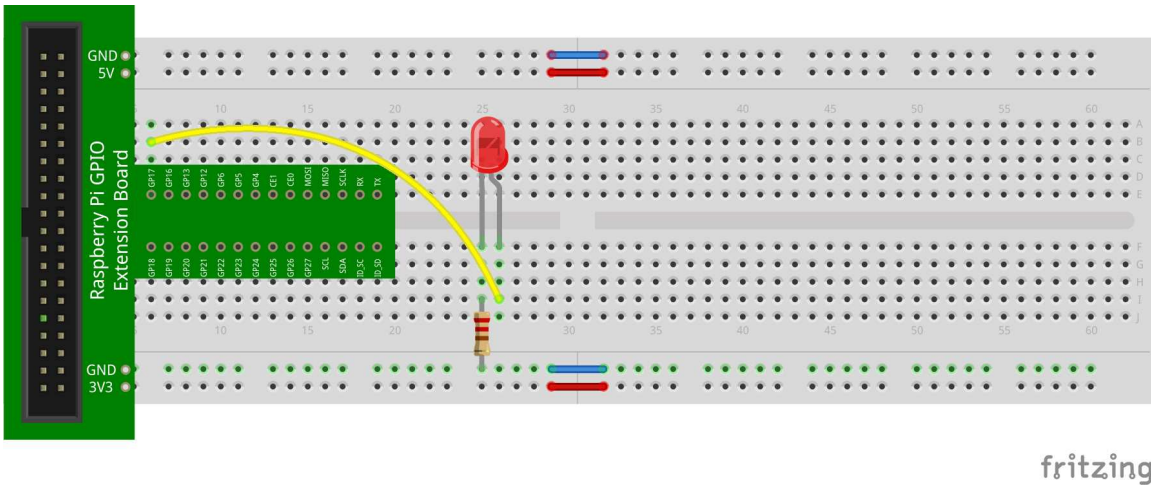
Power is extended to one pin **of both switches** (via the two long red wires). The second pin **of both switches** is connected to the positive side of the LED. Try the circuit. The LED should light when either (or both) switches are closed.

Sensing

The previous circuits in this activity have been entirely external of the RPi. That is, the RPi was only used as a power source (basically, a battery). There are many more GPIO pins than the ones we have used so far (3.3V, 5V, and ground). In fact, many of the GPIO pins can be used to provide sensor input to the RPi. Others can be used to provide output capabilities from the RPi. For example, we can programmatically detect when a switch is closed and trigger an LED to light.

But before we can get to this, we must first discuss how to access and manipulate the GPIO pins on the RPi in Python. Fortunately, Python has a library called RPi.GPIO that can be imported. This library is installed by default on the RPi. Let's start with a simple example: lighting an LED. Construct the following circuit (which slightly differs from the first part of this activity):

Here's one way to layout this circuit:



Note that the yellow wire connects the positive side of the LED to a GPIO pin labeled **GP17** on the GPIO interface. If you have the **black** GPIO interface board, the yellow wire will connect to the same physical pin location, but the pin on the GPIO interface board will be labeled **P0** instead of **GP17**. This is a good time to discuss pin numbering schemes. It turns out 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**. Pins also have a name (e.g., 5V, GND, GPIO.0, etc). Here's a table that cross-references each (and includes names of the pins):

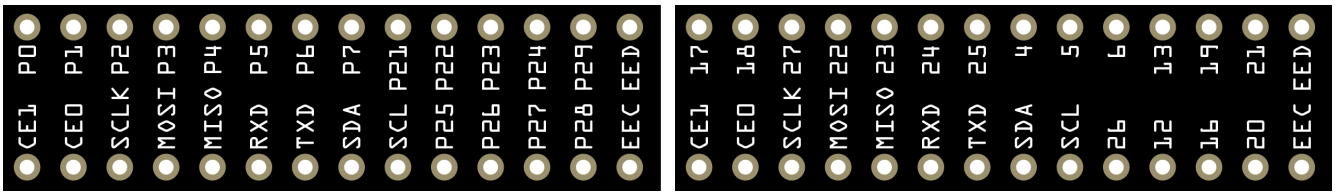
BCM	wPi	Name	Physical		Name	wPi	BCM
		3V3	1	2	5V		
2	8	SDA.1	3	4	5V		
3	9	SCL.1	5	6	GND		
4	7	GPIO.7	7	8	TXD	15	14
		GND	9	10	RXD	16	15
17	0	GPIO.0	11	12	GPIO.1	1	18
27	2	GPIO.2	13	14	GND		
22	3	GPIO.3	15	16	GPIO.4	4	23
		3V3	17	18	GPIO.5	5	24
10	12	MOSI	19	20	GND		
9	13	MISO	21	22	GPIO.6	6	25
11	14	SCLK	23	24	CE0	10	8
		GND	25	26	CE1	11	7
0	30	SDA.0	27	28	SCL.0	31	1
5	21	GPIO.21	29	30	GND		
6	22	GPIO.22	31	32	GPIO.26	26	12
13	23	GPIO.23	33	34	GND		
19	24	GPIO.24	35	36	GPIO.27	27	16
26	25	GPIO.25	37	38	GPIO.28	28	20
		GND	39	40	GPIO.29	29	21

Don't worry about understanding the names of the pins for now (although you may have noticed that they somewhat correlate with the wiringPi numbering scheme). The GPIO pin in the layout diagram above that has the yellow wire connecting to the LED is labeled GP17. The labeling on the green GPIO interface in your kit uses Broadcom (BCM in the table above). GP17 is just BCM pin 17 which cross-

references to wiringPi (wPi) pin 0 and physical pin 11 on the RPi. You may need this reference chart anytime you write Python programs that make use of the GPIO pins.

Note that Python primarily uses the Broadcom (BCM) pin numbering scheme which, thankfully, matches the GPIO interface board! For those of you that have the **black** GPIO interface board, using this chart can easily provide a crossreference from a BCM pin to a wPi one. In this case, BCM pin 17 (**GP17**) matches wPi pin 0 (**P0**). In a Python program, we simply need to refer to BCM pin 17 to match wPi pin 0.

For reference, here's a comparison of the **black** GPIO interface boards labeled with both pin numbering schemes (wPi on the left, and BCM on the right):



Note that the green GPIO interface board is labeled directly using the BCM pin numbering scheme; therefore, no crossreference is needed.

Importing the RPi.GPIO library is a simple as including the following **import** statement (typically done at the beginning of a Python program):

```
import RPi.GPIO as GPIO
```

To refer to GPIO pins using the Broadcom pin layout, set the mode as follows:

```
GPIO.setmode(GPIO.BCM)
```

To turn the LED on, we must first configure pin 17 (again, using the BCM pin layout) to be an **output** pin as follows:

```
GPIO.setup(17, GPIO.OUT)
```

And finally, to turn on the LED:

```
GPIO.output(17, GPIO.HIGH)
```

This turns the pin on by supplying it 3.3V. And that's all there is to turning on an LED! Here's the full program for reference:

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)          # set the pin mode
GPIO.setup(17, GPIO.OUT)        # setup pin 17 as an output pin
GPIO.output(17, GPIO.HIGH)      # 3.3V to the pin (turn on the LED)
```

Turning the LED off can be done as follows (which turns the pin off by supplying it with 0V):

```
GPIO.output(17, GPIO.LOW)       # 0V to the pin (turn off the LED)
```

Did you know?

Instead of using GPIO.HIGH to supply 3.3V to a pin, you can use 1 or True. For example, the following statements are identical (i.e., they produce the same result):

```
GPIO.output(17, GPIO.HIGH)
GPIO.output(17, 1)
GPIO.output(17, True)
```

Likewise, the following statements are identical and supply 0V to a pin:

```
GPIO.output(17, GPIO.LOW)
GPIO.output(17, 0)
GPIO.output(17, False)
```

Now, let's try to blink the LED. This will mean turning the output pin on, waiting some amount of time, turning the output pin off, waiting some amount of time, and so on. We already know how to turn an output pin high and low. We also know how to repeat a task over and over (we can use a while loop!). But we'll need to introduce a small delay so that we can actually see the LED blink. To do this, we can import the *time* library and make use of its `sleep` function:

```
from time import sleep
```

This will allow us to introduce delays. For example, we can introduce a half second delay as follows:

```
sleep(0.5)
```

To blink an LED with a half second delay in between each state of the LED (on or off) and blink the LED *forever*, we can modify our program as follows:

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.OUT)

while (True):
    GPIO.output(17, GPIO.HIGH)
    sleep(0.5)
    GPIO.output(17, GPIO.LOW)
    sleep(0.5)
```

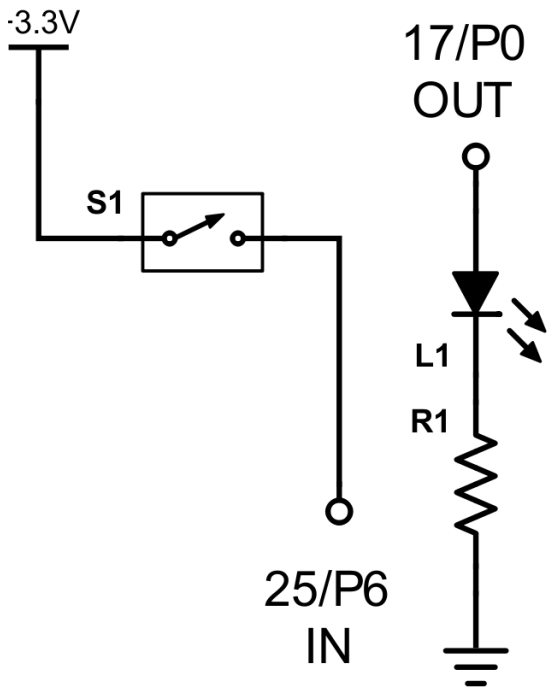
Note that the **while** loop will go on forever (i.e., `while (True)` is never false!). **To stop the program, we can press Ctrl+C**. When doing so, you may notice warnings or errors. This is normal, because our program did not clean up before aborting. In fact, you will most likely also get errors if you try to run the program again (or another program that uses the GPIO pins). These errors are safe to ignore for now. Usually when using GPIO pins, it is recommended to clean them up so that they are reset. We won't worry about this right now.

Often, it is standard practice to assign GPIO pin numbers to meaningful variables. For example, we can assign pin 17 to the variable *led* (since in our program it is used to control an LED). In the end, we can modify our program as follows:

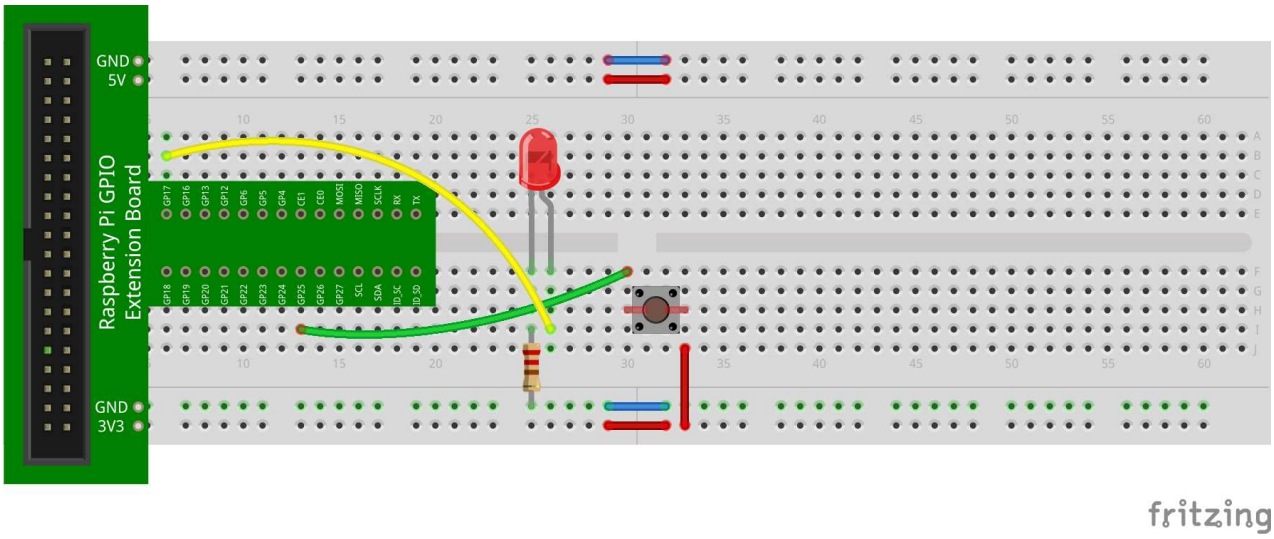
```
import RPi.GPIO as GPIO
from time import sleep

led = 17

GPIO.setmode(GPIO.BCM)
GPIO.setup(led, GPIO.OUT)

while (True):
     GPIO.output(led, GPIO.HIGH)
     sleep(0.5)
     GPIO.output(led, GPIO.LOW)
     sleep(0.5)
```

Adding a switch

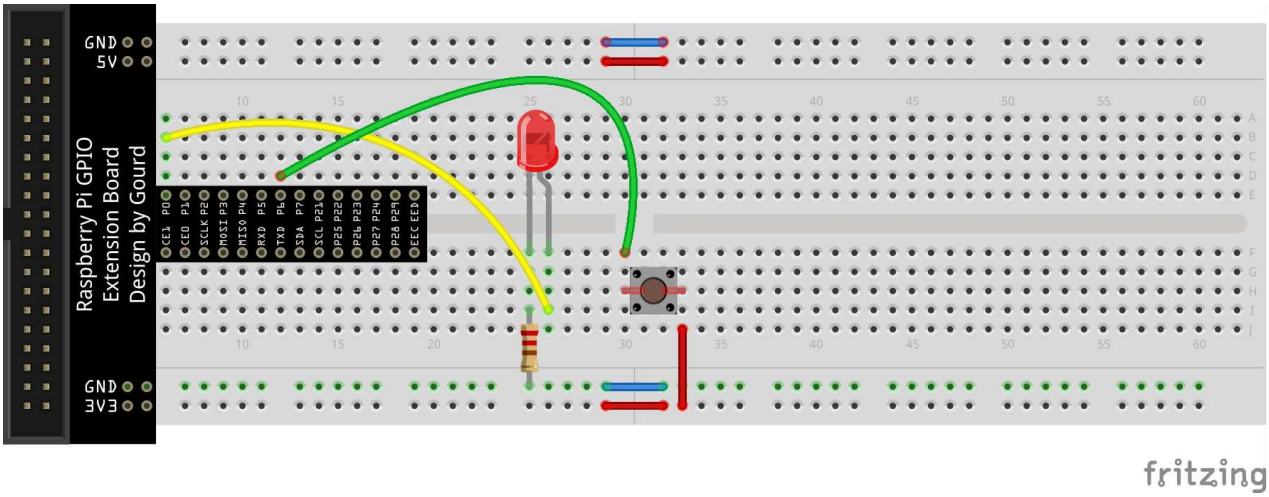
Let's add a switch to the mix. To make this work, we want to programmatically detect the status of the switch (open or closed). If the switch is closed, then the LED should turn on; otherwise, it should remain off. To begin, implement the following circuit:



Here's one way to layout this circuit:



If you have the **black** GPIO interface board, layout the circuit as follows instead:



Again, the yellow wire is connected to GP17 (wPi P0) and the positive side of the LED. The green wire is connected to one side of the switch and to **GP25** (wPi **P6**). The LED portion is unchanged from the last circuit. The only difference is the addition of a switch. One side is connected to +3.3V, and the other is connected to GP25 (wPi P6).

Detecting the state of the switch is not particularly difficult. The pin's state is first setup to be an input pin. While we're at it, we'll set a variable, *button*, to store the number of the pin (like we did with the LED above):

```
button = 25
GPIO.setup(button, GPIO.IN)
```

In Python, input switches can be wired to positive voltage (e.g., +3.3V as in the layout diagram above) or to ground. Let's refer to the case where one pin of the switch is wired to +3.3V and the other to the input pin as **CASE 1**. By default, the input pin should be low. In fact, it should be intentionally **pulled down** to GND to ensure this. If the switch is open, current cannot flow to the input pin. Pressing the

switch closes the circuit and allows +3.3V to flow to the input pin, thereby setting it high. The state change can be read to detect the pushing of the button!

We'll define **CASE 2** to be the case where one pin of the switch is wired to GND and the other to the input pin. In this configuration, the input pin is **pulled up** and actually provides current (but it has nowhere to go if the switch is not closed). Pressing the switch closes the circuit and allows current from the input pin to flow to GND. Again, the state change can be read to detect the pushing of the button.

Since both of these ways of detecting an input are possible in Python, it is standard practice to set an input pin's *default* state (which depends on how it is wired). We do so by either connecting the input pin (internally through our program) to 3.3V or to ground. To connect the pin to 3.3V, the RPi internally uses a **pull-up** resistor (which *pulls* the state of the input pin *up* to 3.3V). To connect the pin to ground, the RPi internally uses a **pull-down** resistor (which *pulls* the state of the input pin *down* to 0V). Specifying a default input pin state can be done as follows:

```
button = 25
# for case 1
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
# for case 2
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Which you choose doesn't matter in most cases. Just make sure that you connect the other side of the switch as appropriate (e.g., to +3.3V if the input pin has a pull-down resistor and is low by default, or to ground if the input pin has a pull-up resistor and is high by default).

Did you know?

You can set multiple GPIO pins either as input or output in one single statement. The method involves providing a list of the GPIO pins to the setup command as follows:

```
out_pins = [17, 18]
in_pins = [22, 27]
GPIO.setup(out_pins, GPIO.OUT)
GPIO.output(in_pins, GPIO.IN)
```

This sets GPIO 17 and 18 as input pins and GPIO 22 and 27 as output pins. In fact, this can also be used to set all of the output pins in the output pin list (GPIO 17 and 18) as either high or low as follows:

```
GPIO.output(out_pins, GPIO.HIGH)
```

Setting GPIO 17 high and GPIO 18 low can be done in one statement as follows:

```
GPIO.output(out_pins, (GPIO.HIGH, GPIO.LOW))
```

Reading the state of an input pin can be done as follows:

```
if (GPIO.input(button) == GPIO.HIGH):
    ...
```

To begin, let's just display a status message that informs us whether the switch is open or closed. Here's the full Python program:

```
import RPi.GPIO as GPIO
```

```
from time import sleep

led = 17
button = 25

GPIO.setmode(GPIO.BCM)
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

while (True):
     if (GPIO.input(button) == GPIO.HIGH):
          print "Closed!"
     else:
          print "Open!"
     sleep(1)
```

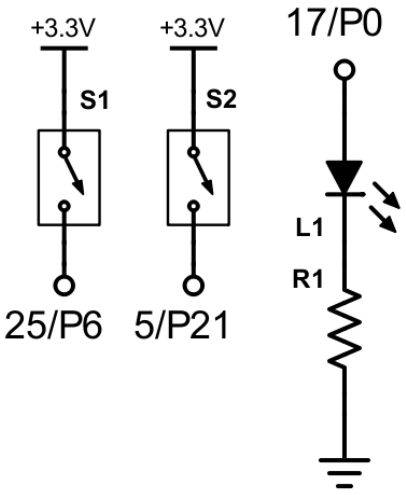
Try running the program. Notice that when the switch is open, "Open!" appears; otherwise, "Closed!" appears. This happens every second. Why?

To *connect* the switch to the LED (i.e., to make the switch control the LED), simply change the statements in the while loop as follows (note that the rest of the program remains unchanged). While we're at it, we can sleep a little less each time to allow the circuit to react faster to changes in the switch state:

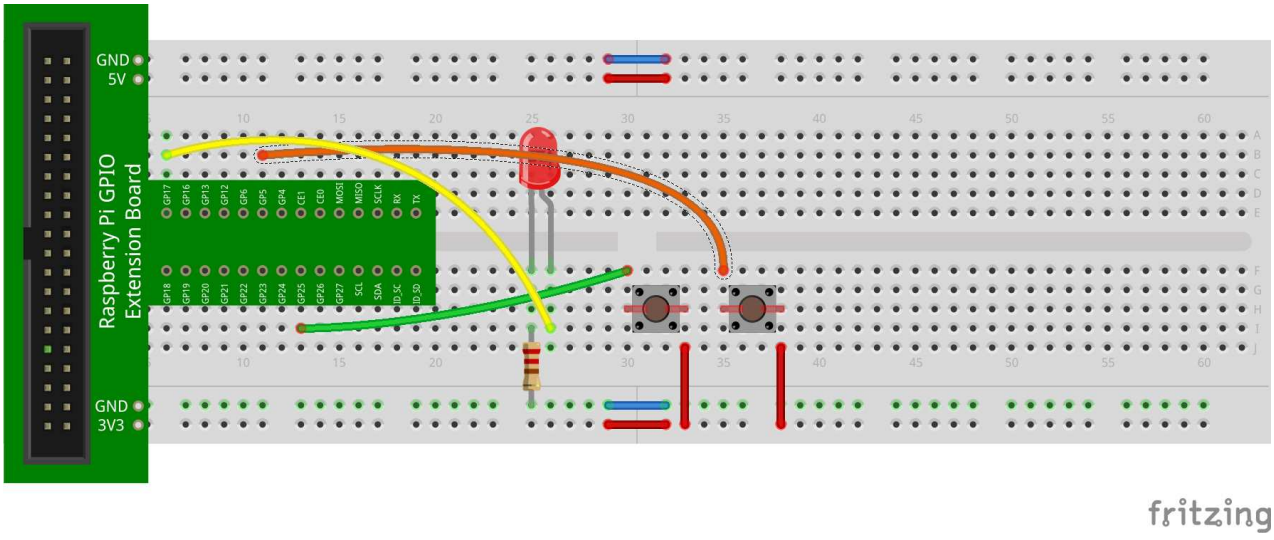
```
while (True):
     if (GPIO.input(button) == GPIO.HIGH):
          GPIO.output(led, GPIO.HIGH)
     else:
          GPIO.output(led, GPIO.LOW)
     sleep(0.1)
```

Detecting an input pin in this way is called **polling**. The input pin is repeatedly polled (checked) for its state. As you can see, this repeats forever and can use a lot of CPU processing time. There are better ways to detect changes in input pins that do not keep the CPU so busy; however, this will work for now.

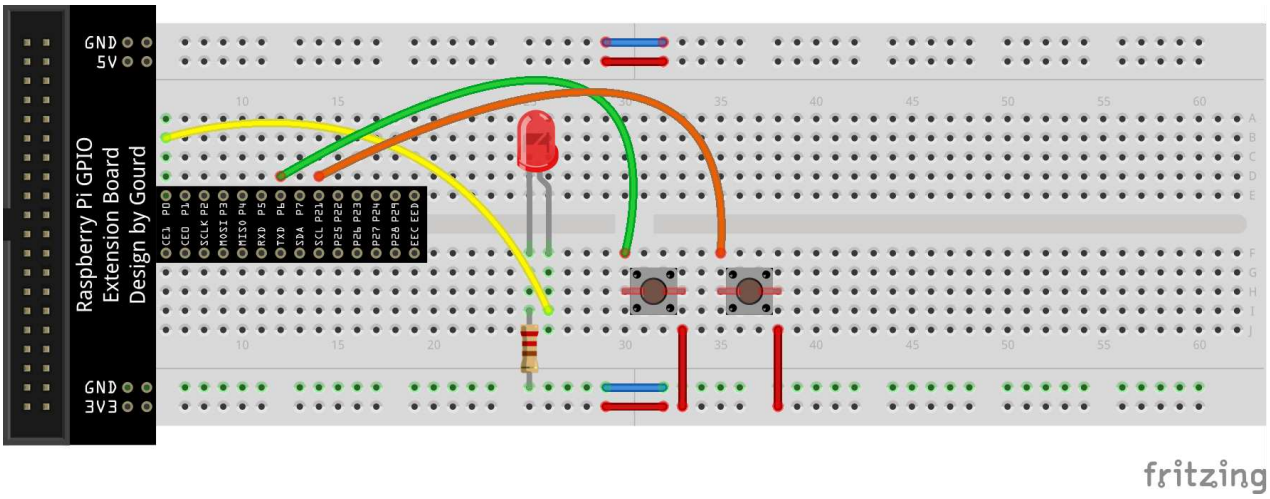
Two switches (to implement *and* and *or*)
Earlier, to manually implement *and* and *or*, two switches had to be wired either in series or parallel. Programmatically doing so precludes this. The logic can be done in Python! Let's try this by first implementing the following circuit:



Here's one way to layout this circuit:



If you have the **black** GPIO interface board, layout the circuit as follows instead:



The only difference in this circuit is the addition of the second switch. It is connected to +3.3V and to **GP5** (wPi **P21**). To implement the functionality of *and* (i.e., replicating two switches in series), we simply need to turn the LED on when both input pins read high. We can do this by implementing the following Python program:

```
import RPi.GPIO as GPIO
from time import sleep

led = 17
button1 = 25
button2 = 5

GPIO.setmode(GPIO.BCM)
GPIO.setup(led, GPIO.OUT)
GPIO.setup(button1, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(button2, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

while (True):
    if (GPIO.input(button1) == GPIO.HIGH and GPIO.input(button2) == GPIO.HIGH):
        GPIO.output(led, GPIO.HIGH)
    else:
        GPIO.output(led, GPIO.LOW)
    sleep(0.1)
```

The logic is actually quite clear: the LED is turned on if both button1 (on BCM pin 25/wPi P6) **and** button2 (on BCM pin 5/wPi P21) are high. Both conditions (on the left and right of the **and** operator) must be true in order for the entire if-statement to be true.

Of course, implementing the *or* gate is just as easy. In fact, there is no need to change the circuit! We simply switch the *and* operator with the *or* operator. The rest of the logic is exactly the same:

```
if (GPIO.input(button1) == GPIO.HIGH or GPIO.input(button2) == GPIO.HIGH):
```

Did you know?

When circuits are continuously toggled (such as when an LED is turned on and off, over and over), we can refer to the portion of time that the circuit is on as a duty cycle. Formally, a **duty cycle** is the percentage of one period in which a signal is active. A period is the time it takes for a signal to complete an on-and-off cycle. In a simple LED circuit, a duty cycle of 50% means that the LED turns on and off for the same amount of time (e.g., the LED turns on for one second, off for one second, and so on). A duty cycle of 25% means that the LED turns on 25% of the time (e.g., the LED turns on for 0.25s, off for 0.75s, and so on).

Homework: Blink!

Implement a single LED, single switch circuit and write a Python program that does the following:

(1) The LED should blink continuously such that it is on for 0.5s and off for 0.5s;
(2) When the switch is pressed (closed), it should change the LED's blink rate so that it is on for 0.1s and off for 0.1s (i.e., it should make the LED blink faster); and
(3) When the switch is released (open), the LED should go back to blinking at the original rate of 0.5s on and 0.5s off.

Submit your Python source file (i.e., the one with a .py extension) through the upload facility on the web site.

Raspberry Pi Activity Assignment: LED the Way

Calculations

Suppose that a circuit has a power source voltage of 9V, and an LED with a forward voltage (i.e., voltage drop) of 2.5V that requires 25mA of current for optimum brightness. In the space below, calculate the resistance of the series resistor required. State the formula used as the basis for your answer, identify all units, and show all work!

In the space below, calculate the power dissipated by the resistor. State the formula used as the basis for your answer, identify all units, and show all work!

In the space below, calculate the power dissipated by the LED. State the formula used as the basis for your answer, identify all units, and show all work!

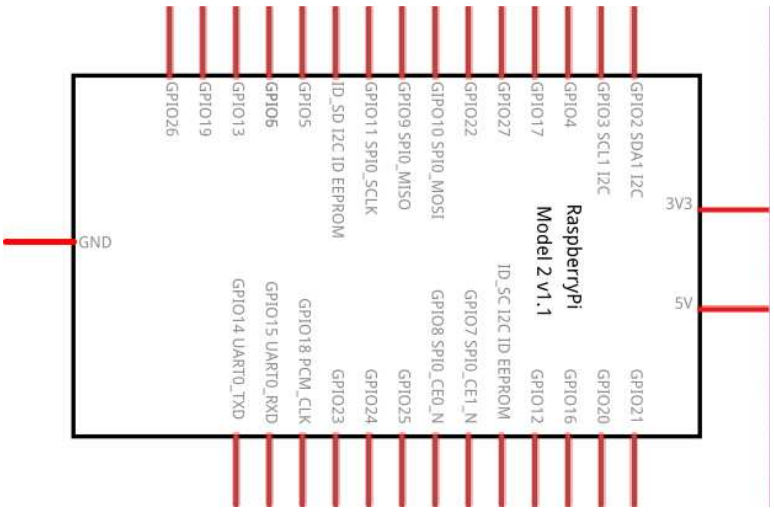
Truth tables
Fill in the truth table for the *and* gate below:

A	B	Z

Fill in the truth table for the *xor* (exclusive *or*) gate below:

A	B	Z

Circuits
Using the figure below, draw the single-switch, single-LED circuit in which the RPi was responsible for detecting the state of the switch (as input) and accordingly affecting the blink rate of the LED (as output):



Raspberry Pi Activity: Ultrasonic Sort

In this activity, you will implement a circuit that contains an ultrasonic distance sensor. After calibrating the sensor, you will use it to detect items at various distances and subsequently sort the item distances using one of the three sorting algorithms that you have learned so far (i.e., bubble sort, selection sort, and insertion sort). You will need the following items:

- Raspberry Pi B v3 with power adapter;
- LCD touchscreen;
- Keyboard and mouse;
- Breadboard;
- GPIO-to-breadboard interface board with ribbon cable;
- Jumper wires provided in your kit;
- 1K Ohm and 2K Ohm resistors (provided to you during the activity); and
- HC-SR04 ultrasonic ranging module (provided to you during the activity).

Connecting the breadboard

Since this activity makes use of an external circuit, you will need to connect the Raspberry Pi to the breadboard through the GPIO interface board. If necessary, refer to the *LED the Way* Raspberry Pi activity for details on how to connect the Raspberry Pi to the breadboard, and for wiring of power and ground across the top and bottom rails of the breadboard. **Make sure that the ribbon cable is not connected backwards!**

Measuring an object's distance

Although there are many different ways that the distance to an object can be measured, this activity makes use of an ultrasonic range module (specifically, the HC-SR04). This "sensor" provides accurate distance measurements from 2cm to 400cm. It includes a transmitter (basically, speakers), a receiver, and control circuitry – so it can easily interface with the RPi!

The HC-SR04 has four pins:

- **VCC**: power (requires 5V)
- **GND**: ground
- **TRIG**: trigger (sends an ultrasonic pulse)
- **ECHO**: echo (transmits the duration of the ultrasonic pulse)

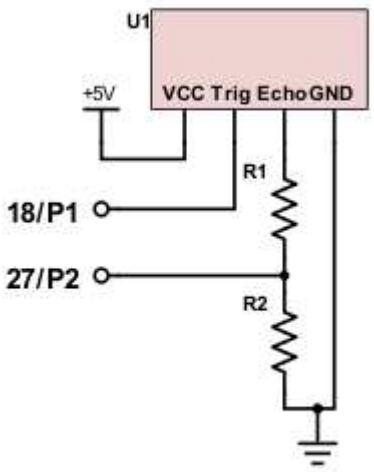
The transmitters emit a high frequency ultrasonic sound (i.e., sound waves that are typically greater than 20 kHz – higher than humans can hear) that bounces (i.e., is reflected) off of any solid object. The idea is that some of the ultrasonic sound bounces off of the solid object, is subsequently detected by the receiver on the sensor, and finally processed by the control circuitry to calculate the time difference between sending and receiving the reflected ultrasonic sound. With a few final calculations (on the RPi), the distance between the sensor and the object can be calculated.

Specifically, the RPi will be used to send an output signal (i.e., set a GPIO output pin high) to the sensor's **TRIG** pin. This will trigger the sensor to send an ultrasonic pulse. This pulse will bounce off of the desired object, allowing the sensor to detect some of the reflected ultrasonic sound waves. The sensor then calculates the time that it took for the pulse to travel from its transmitters to the object and back to its receiver. It then sets its **ECHO** pin high for the duration that it calculated. Since the sensor's **ECHO** pin is connected to a GPIO input pin, the RPi can detect the duration.

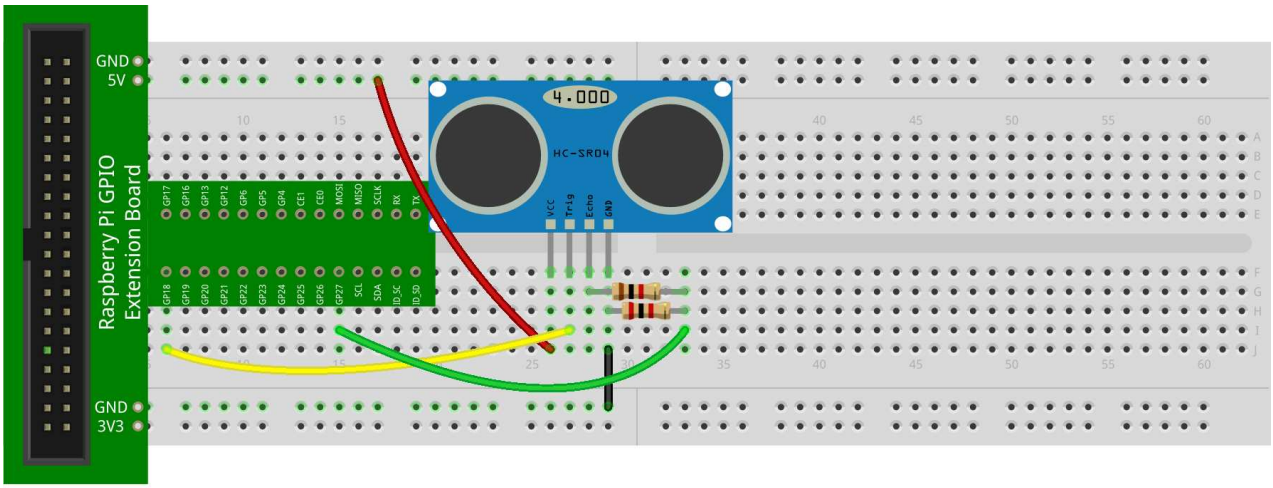
The RPi must then measure how long the GPIO input pin connected to the sensor's **ECHO** pin is set to high by the sensor. As noted, this duration represents how long it took for the sensor's pulse to leave its transmitters, get reflected off of the object, and return back to its receiver. The distance from the sensor to the object can then be calculated.

The circuit

The main goal is to connect the sensor's **VCC** pin to the RPi's 5V source, **GND** to the RPi's ground, **TRIG** to a GPIO output pin (although we'll do so in a different way), and **ECHO** to a GPIO input pin. First, implement the following circuit:

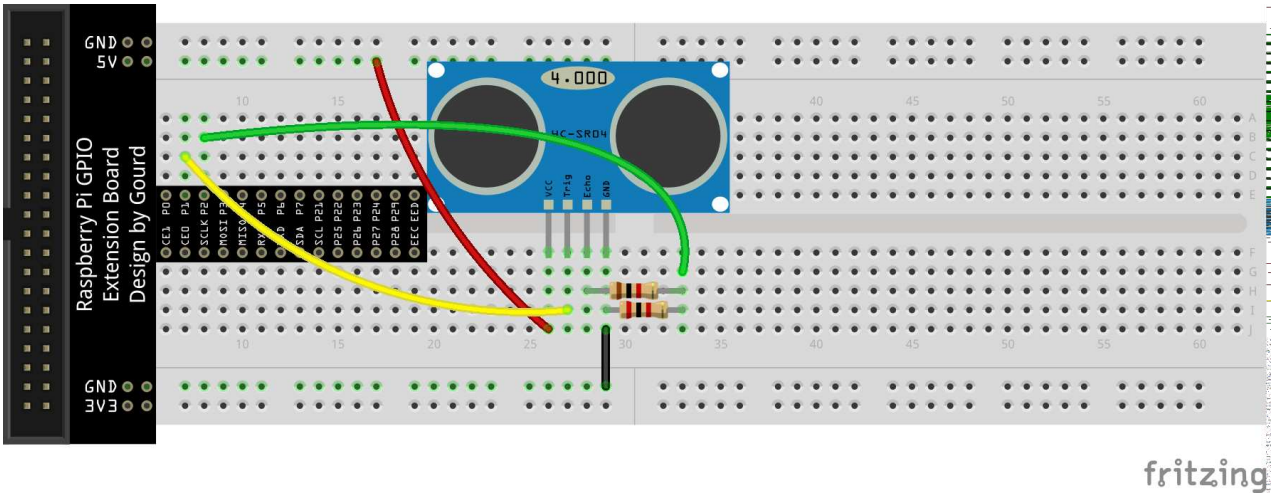


Here's one way to layout the circuit:



Note the use of the two resistors! The sensor's **ECHO** pin is not directly connected to a GPIO pin on the RPi! This will be discussed in more detail later.

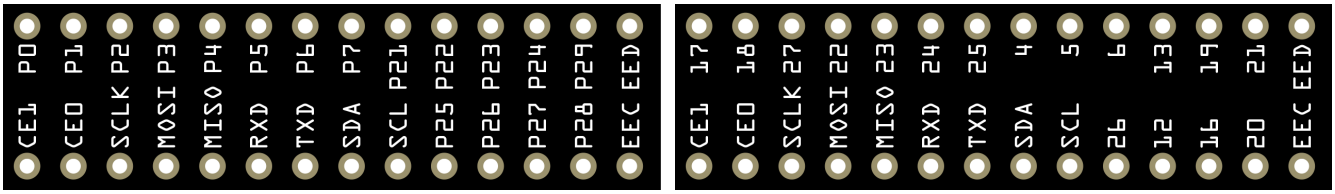
We'll use GP18 for the sensor's **TRIG** pin and GP27 for the sensor's **ECHO** pin. If you have the black GPIO interface board, layout the circuit as follows instead:



Recall that GP18 (on the green GPIO interface board) matches with P1 (on the black GPIO interface board). Similarly, GP27 matches with P2. This was discussed earlier and made reference to the following table:

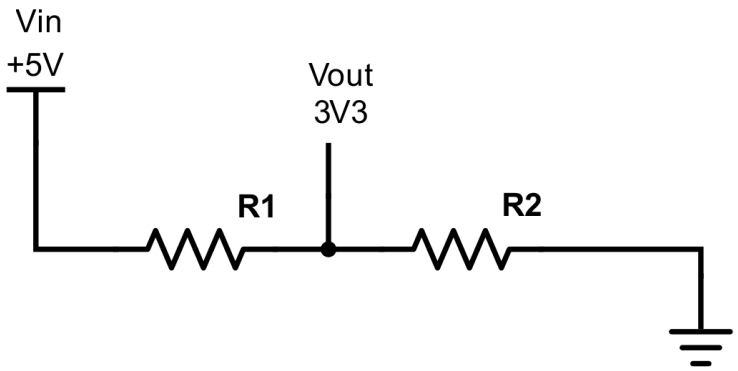
BCM	wPi	Name	Physical		Name	wPi	BCM
		3V3	1	2	5V		
2	8	SDA.1	3	4	5V		
3	9	SCL.1	5	6	GND		
4	7	GPIO.7	7	8	TXD	15	14
		GND	9	10	RXD	16	15
17	0	GPIO.0	11	12	GPIO.1	1	18
27	2	GPIO.2	13	14	GND		
22	3	GPIO.3	15	16	GPIO.4	4	23
		3V3	17	18	GPIO.5	5	24
10	12	MOSI	19	20	GND		
9	13	MISO	21	22	GPIO.6	6	25
11	14	SCLK	23	24	CE0	10	8
		GND	25	26	CE1	11	7
0	30	SDA.0	27	28	SCL.0	31	1
5	21	GPIO.21	29	30	GND		
6	22	GPIO.22	31	32	GPIO.26	26	12
13	23	GPIO.23	33	34	GND		
19	24	GPIO.24	35	36	GPIO.27	27	16
26	25	GPIO.25	37	38	GPIO.28	28	20
		GND	39	40	GPIO.29	29	21

The following simpler cross-reference was also included. It provided a conversion from the pin layout on the black GPIO interface board (on the left) that implements the wiringPI pin layout to the BCM pin layout that the green GPIO interface board uses (on the right):



Therefore, if the **TRIG** pin on the sensor is connected to GP18 on the green GPIO interface board, then it would be connected to P1 on the black GPIO interface board.

You may have noticed that the sensor's **ECHO** pin is actually connected to a 1K Ohm (brown, black, red) resistor. This resistor is connected to both GP27 and to a 2K Ohm (red, black, red) resistor. The second resistor is then connected to GND. This circuit configuration is called a voltage divider. The RPi's input pins can handle 0V to 3.3V. Anything greater than 3.3V could potentially damage the RPi! The HC-SR04 sensor's **ECHO** pin produces 5V when set high because its VCC pin is connected to the RPi's 5V supply. This voltage is too high and must therefore be reduced to 3.3V when connected to an input pin on the RPi. We can use two resistors in series to effectively reduce the voltage. In general, a voltage divider circuit is configured as follows:



To see how this circuit works, we can use Ohm's Law (which you should be familiar with). Let's consider +5V in the circuit as the input voltage, V_{in}, and +3.3V as the desired output voltage, V_{out}. There are two resistors in series, R_1 and R_2. Using Ohm's Law, we can calculate the current flowing through the entire circuit:

$$\begin{aligned} V &= IR \\ V_{\text{in}} &= I(R_1+R_2) \\ I &= \frac{V_{\text{in}}}{R_1+R_2} \end{aligned}$$

Now that we know the current flowing through the entire circuit (in general terms), we can calculate the output voltage, V_{out}, based on this (note that only R_2 exists from V_{out} to ground):

$$\begin{aligned} V &= IR \\ V_{\text{out}} &= IR_2 \\ V_{\text{out}} &= V_{\text{in}}\frac{R_2}{R_1+R_2} \end{aligned}$$

To properly reduce the voltage, the resistor values required in the circuit must be calculated. The easiest way to do this is to set one of the resistors to a known value and calculate the other. We can therefore solve for R_2 as follows:

$$\begin{aligned} V_{\text{out}} &= V_{\text{in}}\frac{R_2}{R_1+R_2} \\ \frac{V_{\text{out}}}{V_{\text{in}}} &= \frac{R_2}{R_1+R_2} \\ V_{\text{out}}(R_1+R_2) &= V_{\text{in}}R_2 \\ V_{\text{out}}R_1+V_{\text{out}}R_2 &= V_{\text{in}}R_2 \\ V_{\text{in}}R_2-V_{\text{out}}R_2 &= V_{\text{out}}R_1 \\ R_2(V_{\text{in}}-V_{\text{out}}) &= V_{\text{out}}R_1 \\ R_2 &= \frac{V_{\text{out}}R_1}{V_{\text{in}}-V_{\text{out}}} \end{aligned}$$

Let's set R_1 to 1K Ohm. We now have the following: V_{in} is +5V, the desired V_{out} is +3.3V, and R_1 is 1K Ohm. We can calculate the value of R_2:

$$R_2 = \frac{3.3\,\text{V}(1000\,\text{Ohms})}{5\,\text{V}-3.3\,\text{V}}$$

$$R_2 = \frac{3300 \text{ V Ohms}}{1.7 \text{ V}}$$

$$R_2 = 1941.18 \text{ Ohms}$$

Therefore, a 2K Ohm resistor for R_2 should work just fine! In the layout diagrams above, you can see that the sensor's **ECHO** pin is connected to the 1K Ohm and 2K Ohm resistors in series to GND. The RPi's GPIO input pin is connected in between the two resistors, where the voltage (V_{out}) has been appropriately reduced from 5V to 3.3V.

You may wonder why we don't simply connect the sensor's VCC pin directly to 3.3V, thereby ensuring that its **ECHO** pin will also produce 3.3V. Unfortunately, the sensor is not stable at 3.3V. It requires 5V to sense properly.

Positioning
To properly measure the distance between the sensor and various objects, make sure to orient the sensor so that its transmitters are pointing toward the objects. This may change the orientation of the pins when compared to the layout diagrams above. Make sure that the sensor's pins are properly connected as specified. Also, the breadboard will need to be relatively flat so that the sensor is level resulting in ultrasonic pulses that are sent directly toward the objects.

Programming
The next task is to layout a Python program to accomplish the desired task of measuring the distance of objects using the HC-SR04 sensor. Let's take an incremental approach to accomplish this, developing a little at a time.

It is important to note that the sensor is not perfect. That is, it may be slightly off in its measurements. Therefore, it will need to be calibrated before taking distance measurements. Technically, the sensor itself is not directly calibrated. Instead, several distance measurements to an object that is a known distance away are taken. The average of these is used to determine a correction factor that is subsequently applied to calculations that involve the sensor's distance measurements. This will be discussed in more detail later.

Let's begin with the first iteration of the program, creating a layout that includes general initialization instructions and stubs for any functions:

```
import RPi.GPIO as GPIO
from time import sleep, time

# constants
DEBUG = False                 # debug mode?

# set the RPi to the Broadcom pin layout
GPIO.setmode(GPIO.BCM)

# GPIO pins
TRIG = 18                     # the sensor's TRIG pin
ECHO = 27                     # the sensor's ECHO pin
```

```
GPIO.setup(TRIG, GPIO.OUT)    # TRIG is an output
GPIO.setup(ECHO, GPIO.IN)     # ECHO is an input

# calibrates the sensor
# technically, it returns a correction factor to use in our
#  calculations
def calibrate():
     pass

# uses the sensor to calculate the distance to an object
def getDistance():
     pass

########
# MAIN #
########
# first, allow the sensor to settle for a bit

# next, calibrate the sensor

# then, measure

# finally, cleanup the GPIO pins
```

First, the required libraries are imported. The time library's *sleep* and *time* functions will be used to implement various required timings and delays later.

It is usually good programming practice to provide meaningful state information while a program is running. This is useful when debugging; however, such information is not necessary (nor desired) once everything is in working order. Typically, a debugging variable (as a boolean) is declared (and set to true when in debug mode or false otherwise). State information throughout the code can be easily displayed when in debug mode via if statements. For example:

```
if (DEBUG):
     print "Some useful bit of information."
```

Next, the pin mode is set to the Broadcom pin layout, and the sensor's **TRIG** and **ECHO** GPIO pin numbers and IO settings are defined.

Two functions are then defined (simply *stubbed* out at this time). One will calibrate the sensor, generating (and returning) the correction factor; the other will use the sensor combined with the correction factor to measure and calculate the distance from the sensor to an object.

Finally, the main part of the program is laid out. The process is to first allow the sensor to settle. This is done to ensure that, initially, the sensor is not accidentally triggered. Next, the sensor will be calibrated so that the calculated distance measurements are accurate. The program will eventually support taking multiple measurements through a while loop. When taking measurements, the user will be prompted

whether to take another measurement (or quit) each time. Finally, the GPIO pins will be reset before the program terminates.

Now that the first iteration is complete, test it to make sure that it works. Note that there should be no output at this time.

Settling and calibrating the sensor

Next, let's work on settling and calibrating the sensor. Settling the sensor is quite simple: set the sensor's **TRIG** pin to low and wait a short while to ensure that it isn't triggered to send an ultrasonic pulse. Modify the main part of the program as follows (note that added statements are highlighted):

```
# first, allow the sensor to settle for a bit
print "Waiting for sensor to settle ({}s)...".format(SETTLE_TIME)
GPIO.output(TRIG, GPIO.LOW)
sleep(SETTLE_TIME)
```

Of course, the constant SETTLE_TIME must be defined (note that 2s is adequate for the settling time). Do so in the constants section at the top of the program:

```
# constants
DEBUG = False                   # debug mode?

SETTLE_TIME = 2                 # seconds to let the sensor settle
```

Next, let's calibrate the sensor. First, the call to the *calibrate* function (that returns the correction factor) in the main part of the program:

```
# next, calibrate the sensor
correction_factor = calibrate()
```

Since the *calibrate* function will make use of several constants, let's add them to the constants section as well:

```
SETTLE_TIME = 2                 # seconds to let the sensor settle
CALIBRATIONS = 5                # number of calibration
                                #  measurements to take
CALIBRATION_DELAY = 1           # seconds to delay in between
                                #  calibration measurements
```

The constant CALIBRATIONS specifies the number of calibration measurements to take (five is adequate). The constant CALIBRATION_DELAY specifies a delay in between each calibration measurement. It is good to delay about 1s in between triggers to the sensor to take more measurements.

Now, implement the *calibrate* function:

```
# calibrates the sensor
# technically, it returns a correction factor to use in our
#  calculations
def calibrate():
    print "Calibrating..."
    # prompt the user for an object's known distance
    print "-Place the sensor a measured distance away from an \
```

```
        object."
known_distance = input("-What is the measured distance \
                        (cm)? ")

# measure the distance to the object with the sensor
# do this several times and get an average
print "-Getting calibration measurements..."
distance_avg = 0
for i in range(CALIBRATIONS):
    distance = getDistance()
    if (DEBUG):
        print "--Got {}cm".format(distance)
    # keep a running sum
    distance_avg += distance
    # delay a short time before using the sensor again
    sleep(CALIBRATION_DELAY)
# calculate the average of the distances
distance_avg /= CALIBRATIONS
if (DEBUG):
    print "--Average is {}cm".format(distance_avg)

# calculate the correction factor
correction_factor = known_distance / distance_avg
if (DEBUG):
    print "--Correction factor is \
          {}".format(correction_factor)

print "Done."
print

return correction_factor
```

The *calibrate* function firsts asks the user for the known distance to an object that is in front of the sensor. It then triggers the sensor to send an ultrasonic pulse to this object. It subsequently drives its **ECHO** pin high a duration equal to the amount of time the pulse took to get from the sensor to the object – and back again. This process repeats a number of times, after which the average distance is then calculated. Since the sensor is not perfect, a correction factor (the ratio of the known distance to the average distance) is also calculated. This correction factor can be used in all subsequent distance calculations.

Using the sensor to measure distances

Finally, let's work on actually triggering the sensor to take measurements of objects. First, let's implement the *getDistance* function. We'll need a few constants for this:

```
# constants
...
TRIGGER_TIME = 0.00001          # seconds needed to trigger the
                                #  sensor (to get a measurement)
SPEED_OF_SOUND = 343            # speed of sound in m/s
```

The sensor only measures the time it takes for an ultrasonic pulse to get from its transmitters to an object – and back to its receiver. The distance to the object must therefore be calculated. You should be familiar with the method to calculate distance from speed and time:

distance = speed * time

For example, if you are running at 18 km/hour (~11.18 miles/hour) and run for 1.5 hours, you will have covered 27 km (16.77 miles):

distance = 18 km/hour * 1.5 hours

distance = 27 km

The speed of sound is known to be 343 meters/second, and the duration (time) of the ultrasonic pulse will be obtained from the sensor by monitoring its **ECHO** pin. The distance that the ultrasonic pulse traveled (i.e., from the sensor to the object – and back again) can then be calculated using this formula.

Note that the calculation provides the total distance that the ultrasonic pulse traveled (i.e., from the sensor to the object, and from the object back to the sensor). Since the distance from the sensor to the object is all that is desired, then the distance that the ultrasonic pulse traveled from the object back to the sensor is not necessary. Therefore, the total calculated distance must divided by two!

The end result is in meters (since the speed of sound was specified in this manner). Since the objects that is placed will only be centimeters away, it will be more meaningful to convert the distance from meters to centimeters. Obviously, we can do so by multiplying the distance by 100 (since there are 100 centimeters in a meter).

Did you know?

Although the speed of sound is accepted to be 343 m/s, it is actually affected by the air temperature. Technically, it is 343 m/s at 0ºC. In general, we can precisely calculate the speed of sound as follows: 331.3 + (0.606 * temperature in ºC). For an air temperature of 73ºF (22.78ºC), for example, the actual speed of sound is: 331.3 + (0.606 * 22.78) = 331.3 + 13.81 = 345.1 m/s.

In this activity, an accurate speed of sound isn't necessary, as any differences in the measurements will be inconsequential. Moreover, perfect distance measurements are not required for the sorting portion of this activity.

Let's now implement the *getDistance* function:

```
# uses the sensor to calculate the distance to an object
def getDistance():
    # trigger the sensor by setting it high for a short time and
    #  then setting it low
    GPIO.output(TRIG, GPIO.HIGH)
    sleep(TRIGGER_TIME)
    GPIO.output(TRIG, GPIO.LOW)
```

```
    # wait for the ECHO pin to read high
    # once the ECHO pin is high, the start time is set
    # once the ECHO pin is low again, the end time is set
    while (GPIO.input(ECHO) == GPIO.LOW):
        start = time()
    while (GPIO.input(ECHO) == GPIO.HIGH):
        end = time()

    # calculate the duration that the ECHO pin was high
    # this is how long the pulse took to get from the sensor to
    #  the object -- and back again
    duration = end - start
    # calculate the total distance that the pulse traveled by
    #  factoring in the speed of sound (m/s)
    distance = duration * SPEED_OF_SOUND
    # the distance from the sensor to the object is half of the
    #  total distance traveled
    distance /= 2
    # convert from meters to centimeters
    distance *= 100

    return distance
```

The *getDistance* function first triggers the sensor to take a measurement. It does this by briefly setting the sensor's **TRIG** pin high. The sensor then sends an ultrasonic pulse, measuring how long the pulse takes to get from its transmitters to the object, and back to its receiver. While this happens, the sensor's **ECHO** pin is monitored. As the sensor is taking a measurement, its **ECHO** pin is low. The idea is to repeatedly set a *start* time while the **ECHO** pin is low. Once the **ECHO** pin is high, the last set start time set will be used (which occurs at the moment when the **ECHO** pin transitions from low to high).

The sensor then sets its **ECHO** pin high for the duration of the ultrasonic pulse. By repeatedly setting an *end* time while the **ECHO** pin is high, the last set end time (which occurs at the moment when the **ECHO** pin transitions from high to low) can be used to calculate the duration that the **ECHO** pin was high.

The next step is to calculate the distance as described above: multiply the duration of the pulse by the speed of sound. Since this results in the total distance that the ultrasonic pulse traveled, dividing in half represents the distance from the sensor to the object. Finally, the distance is converted from meters to centimeters.

Let's now actually use the sensor to take measurements. Modify the main part of the program as follows:

```
    # then, measure
    raw_input("Press enter to begin...")
    print "Getting measurements:"
    while (True):
```

```
# get the distance to an object and correct it with the
#  correction factor
print "-Measuring..."
distance = getDistance() * correction_factor
sleep(1)

# and round to four decimal places
distance = round(distance, 4)

# display the distance measured/calculated
print "--Distance measured: {}cm".format(distance)

# prompt for another measurement
i = raw_input("--Get another measurement (Y/n)? ")
# stop measuring if desired
if (not i in [ "y", "Y", "yes", "Yes", "YES", "" ]):
     break
```

For context, the main part of the program first settles the sensor. It then "calibrates" it, generating a correction factor to use in further calculations. The newly added code then describes how to actually measure and calculate an object's distance.

First, to make sure that the user is ready to take measurements, the program is paused until the enter key is pressed. Next, measurements are repeatedly taken until the user no longer wishes to do so. Therefore, this part is contained within a while loop. A single object distance is then measured and calculated by calling the *getDistance* function and applying the correction factor. The result is then rounded to four decimal places (to the right of the decimal point) and displayed. Lastly, the user is asked if taking another measurement is desired. If not, the while loop is exited.

The final if statement in the while loop is interesting. After prompting the user to get another measurement, the response is compared to various strings in a list using the membership operator *in*. The strings in the list represent various forms of a "yes" response (i.e., the user wishes to take another measurement). For simplicity, pressing enter (i.e., entering nothing) is also considered a "yes" response. If the provided input is not in the list containing the various capitalizations of "yes", a "no" response is interpreted, thereby exiting the while loop.

Finally, it is good practice to reset the RPi's GPIO pins. Add the following to the main part of the program:

```
# finally, cleanup the GPIO pins
print "Done."
GPIO.cleanup()
```

That's it! For completeness, here's the program in its entirety:

```
import RPi.GPIO as GPIO
from time import sleep, time

# constants
```

```
DEBUG = False            # debug mode?

SETTLE_TIME = 2          # seconds to let the sensor settle
CALIBRATIONS = 5         # number of calibration measurements to
                         #  take
CALIBRATION_DELAY = 1    # seconds to delay in between
                         #  calibration measurements
TRIGGER_TIME = 0.00001   # seconds needed to trigger the sensor
                         #  (to get a measurement)
SPEED_OF_SOUND = 343     # speed of sound in m/s

# set the RPi to the Broadcom pin layout
GPIO.setmode(GPIO.BCM)

# GPIO pins
TRIG = 18           # the sensor's TRIG pin
ECHO = 27           # the sensor's ECHO pin

GPIO.setup(TRIG, GPIO.OUT)    # TRIG is an output
GPIO.setup(ECHO, GPIO.IN)     # ECHO is an input

# calibrates the sensor
# technically, it returns a correction factor to use in our
#  calculations
def calibrate():
     print "Calibrating..."
     # prompt the user for an object's known distance
     print "-Place the sensor a measured distance away from an \
            object."
     known_distance = input("-What is the measured distance \
                            (cm)? ")

     # measure the distance to the object with the sensor
     # do this several times and get an average
     print "-Getting calibration measurements..."
     distance_avg = 0
     for i in range(CALIBRATIONS):
          distance = getDistance()
          if (DEBUG):
               print "--Got {}cm".format(distance)
          # keep a running sum
          distance_avg += distance
          # delay a short time before using the sensor again
          sleep(CALIBRATION_DELAY)
     # calculate the average of the distances
     distance_avg /= CALIBRATIONS
     if (DEBUG):
          print "--Average is {}cm".format(distance_avg)
```

```
        # calculate the correction factor
        correction_factor = known_distance / distance_avg
        if (DEBUG):
             print "--Correction factor is \
                    {}".format(correction_factor)

        print "Done."
        print

        return correction_factor

    # uses the sensor to calculate the distance to an object
    def getDistance():
        # trigger the sensor by setting it high for a short time and
        #  then setting it low
        GPIO.output(TRIG, GPIO.HIGH)
        sleep(TRIGGER_TIME)
        GPIO.output(TRIG, GPIO.LOW)

        # wait for the ECHO pin to read high
        # once the ECHO pin is high, the start time is set
        # once the ECHO pin is low again, the end time is set
        while (GPIO.input(ECHO) == GPIO.LOW):
             start = time()
        while (GPIO.input(ECHO) == GPIO.HIGH):
             end = time()

        # calculate the duration that the ECHO pin was high
        # this is how long the pulse took to get from the sensor to
        #  the object -- and back again
        duration = end - start
        # calculate the total distance that the pulse traveled by
        #  factoring in the speed of sound (m/s)
        distance = duration * SPEED_OF_SOUND
        # the distance from the sensor to the object is half of the
        #  total distance traveled
        distance /= 2
        # convert from meters to centimeters
        distance *= 100

        return distance

    ########
    # MAIN #
    ########
    # first, allow the sensor to settle for a bit
    print "Waiting for sensor to settle ({}s)...".format(SETTLE_TIME)
```

```
GPIO.output(TRIG, GPIO.LOW)
sleep(SETTLE_TIME)

# next, calibrate the sensor
correction_factor = calibrate()

# then, measure
raw_input("Press enter to begin...")
print "Getting measurements:"
while (True):
     # get the distance to an object and correct it with the
     #  correction factor
     print "-Measuring..."
     distance = getDistance() * correction_factor
     sleep(1)

     # and round to four decimal places
     distance = round(distance, 4)

     # display the distance measured/calculated
     print "--Distance measured: {}cm".format(distance)

     # prompt for another measurement
     i = raw_input("--Get another measurement (Y/n)? ")
     # stop measuring if desired
     if (not i in [ "y", "Y", "yes", "Yes", "YES", "" ]):
          break

# finally, cleanup the GPIO pins
print "Done."
GPIO.cleanup()
```

The final task of sorting the measured/calculated distances is **left to you**. To do so, you should first determine the answers to the following questions:

- How will the various distances be stored (considering that there could be an arbitrary number of them)?
- Where (in the code) should the storing mechanism(s) be initialized (if necessary)?
- Where (in the code) should the distances be stored?
- What sorting algorithm will be used to sort the distances? You know of several sorting algorithms. Which will you use?
- Where (in the code) will the distances be sorted?
- How and where will the unsorted and sorted distances be displayed?

Homework: Ultrasonic Sort

Extend the program in this activity so that it properly stores and sorts the distances measured/calculated. Feel free to implement any of the sorting algorithms that you have learned in this curriculum so far. Your program should first display the distances in the order that they were measured/calculated (i.e.,

unsorted). It should then sort the distances and subsequently display them again, this time in sorted order. Make sure to represent the distances with a precision of four decimal places (to the right of the decimal point).

It is highly recommended that you place the sorting algorithm that you select in its own function that is appropriately called in your program. In fact, not doing so would be considered quite inefficient (and will result in points being deducted).

Here's sample output of a properly implemented program. Of course, the known distance and distance values measured/calculated are unique to this particular run of the program. User input is highlighted in red:

```
Waiting for sensor to settle (2s)...
Calibrating...
-Place the sensor a measured distance away from an object.
-What is the measured distance (cm)? 20.5
-Getting calibration measurements...
Done.

Press enter to begin...
Getting measurements:
-Measuring...
--Distance measured: 20.5464cm
--Get another measurement (Y/n)? y
-Measuring...
--Distance measured: 98.1243cm
--Get another measurement (Y/n)? y
-Measuring...
--Distance measured: 41.3557cm
--Get another measurement (Y/n)? y
-Measuring...
--Distance measured: 19.9691cm
--Get another measurement (Y/n)? y
-Measuring...
--Distance measured: 5.0077cm
--Get another measurement (Y/n)? n
Done.

Unsorted measurements:
[20.5464, 98.1243, 41.3557, 19.9691, 5.0077]
Sorted measurements:
[5.0077, 19.9691, 20.5464, 41.3557, 98.1243]
```

You are to submit your Python source code only (as a .py file) through the upload facility on the web site.