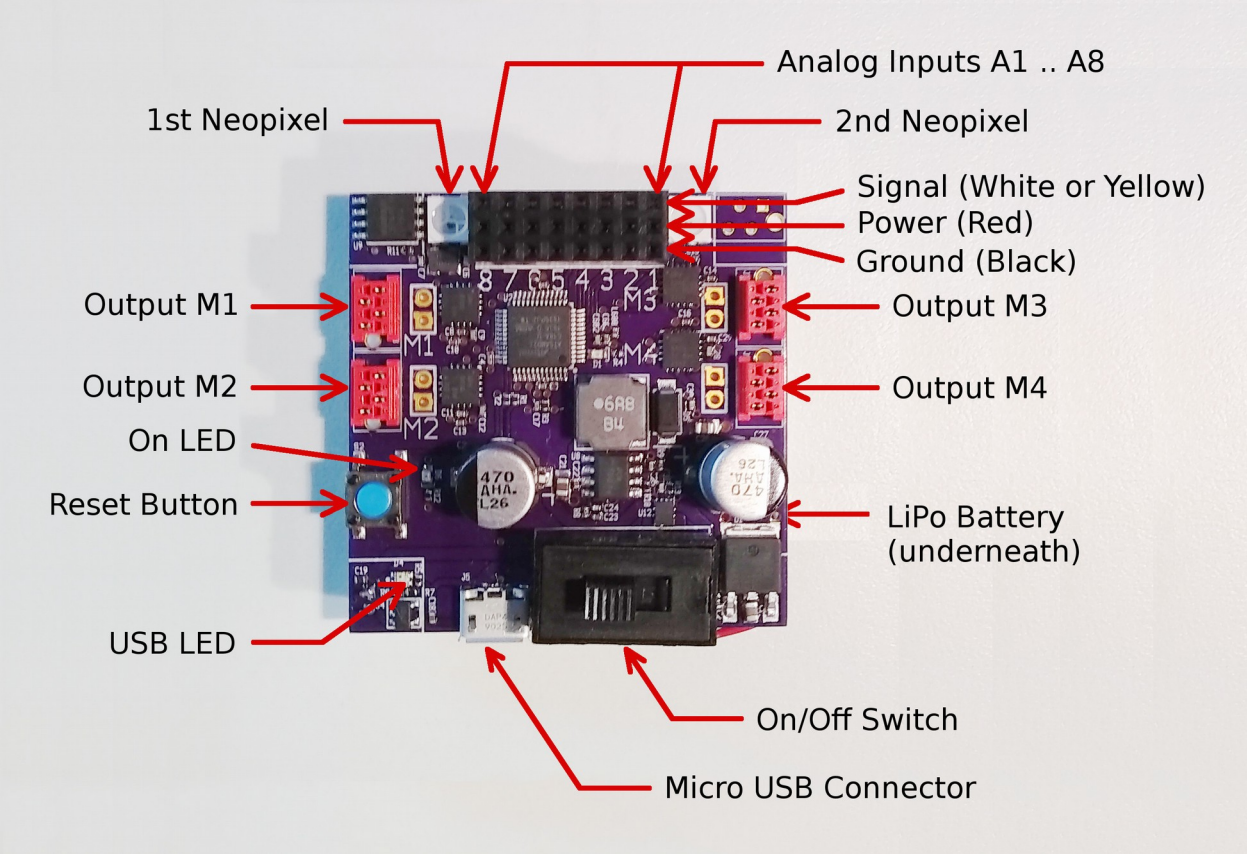
Getting Started With Snek on a Snekboard

Snek is a subset of the Python programming language optimized for small robotics projects. This means that most of what you learn here about writing programs will work in the wide world of Python beyond this class. Snek is written to run on a snekboard designed to control our robots. It has special features for this, so lets take a look at a snekboard.

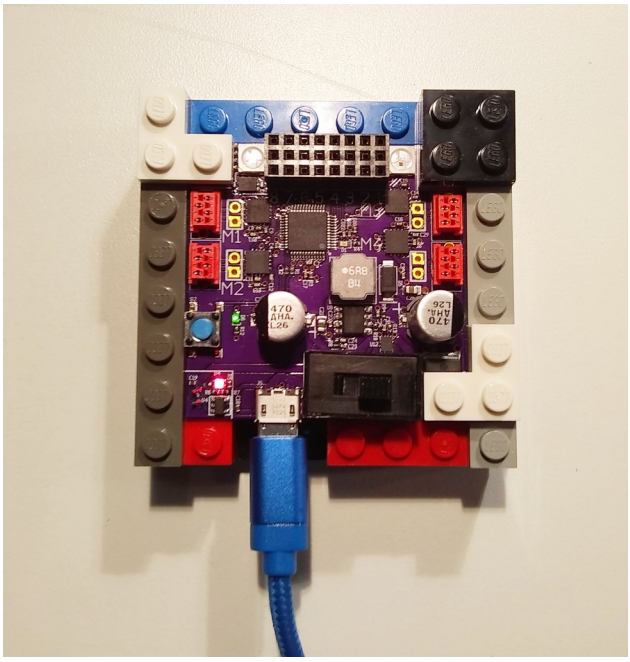
First Look at a SnekBoard

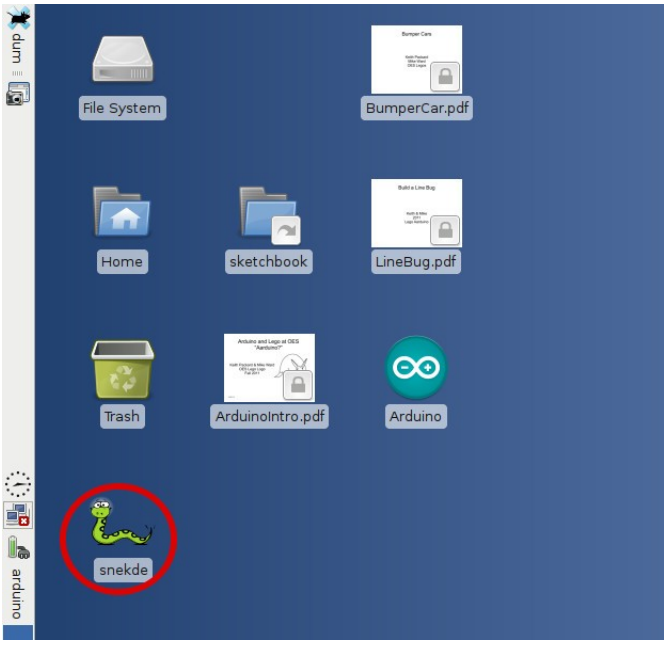


Notice that the snekboard has:

- A lithium-polymer (LiPo) **battery** so it can run with no USB connection.
- A **green LED** that is lit when the board is on.
- A **USB LED** that is:
  - Unlit when there is no power connection.
  - Red when charging.
  - Green when the battery is charged.
- A **reset button** to restart the snekboard (starting the program in memory).
- **8 analog inputs A1 through A8** that return values from 0 to 1.
- **4 outputs M1 thought M4**, each with two directions (left or right) and power from 0 to 1.
- **2 neopixels** that combine a red, green, and blue LED each of which can be lit with a power from 0 to 1.

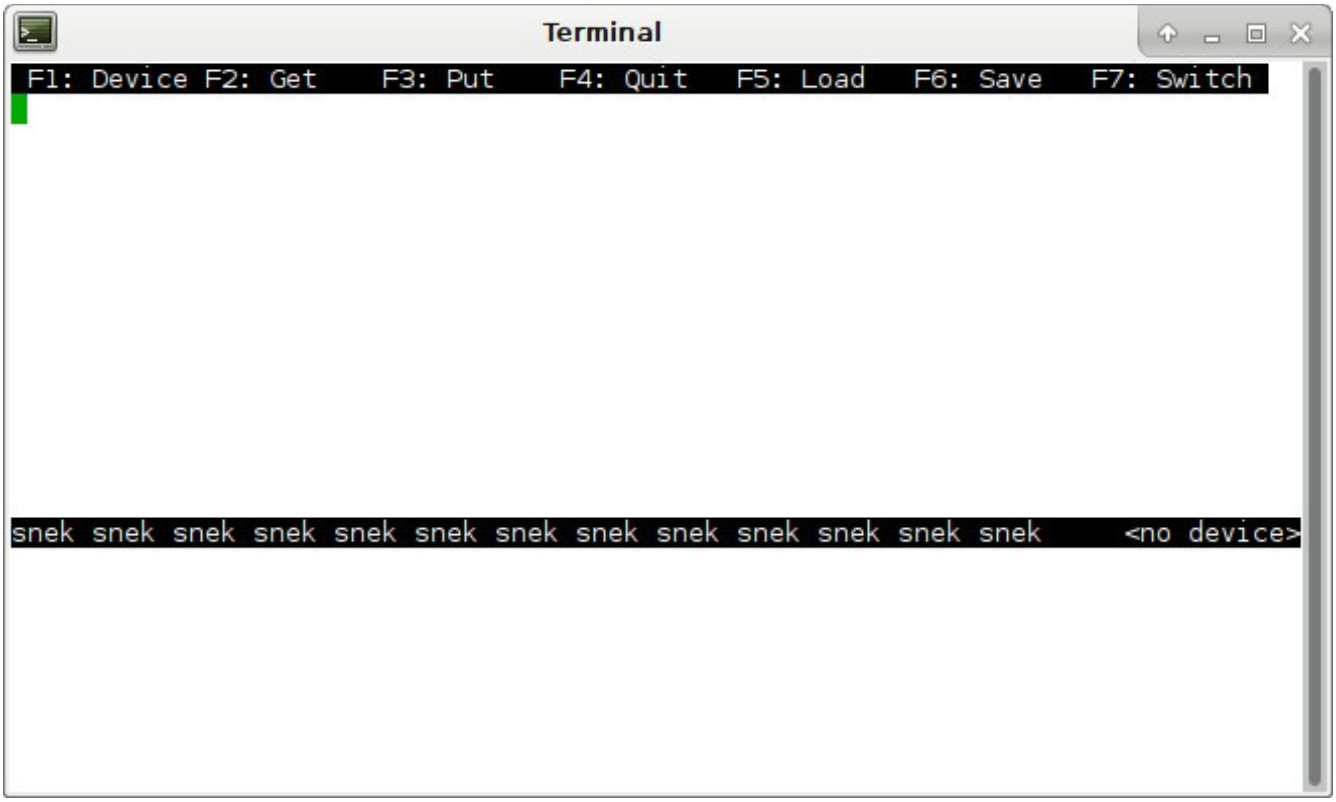
Here is what it looks like when in its Lego case, connected to a laptop, and charging. And here is the Icon for the snek interface (circled in red):





Snek Development Environment (SnekDE)

Once you click the snek icon, the interface opens and looks like this:



Notice that:

- The interface opens with the cursor in the top window. This is the **program editor**. The bottom window is the **interpreter** where you can enter snek expressions that are executed by the snekboard.
- The black line at the top of the interpreter window has `<no device>` at the far right indicating the interface has not connected to a **snek device**.

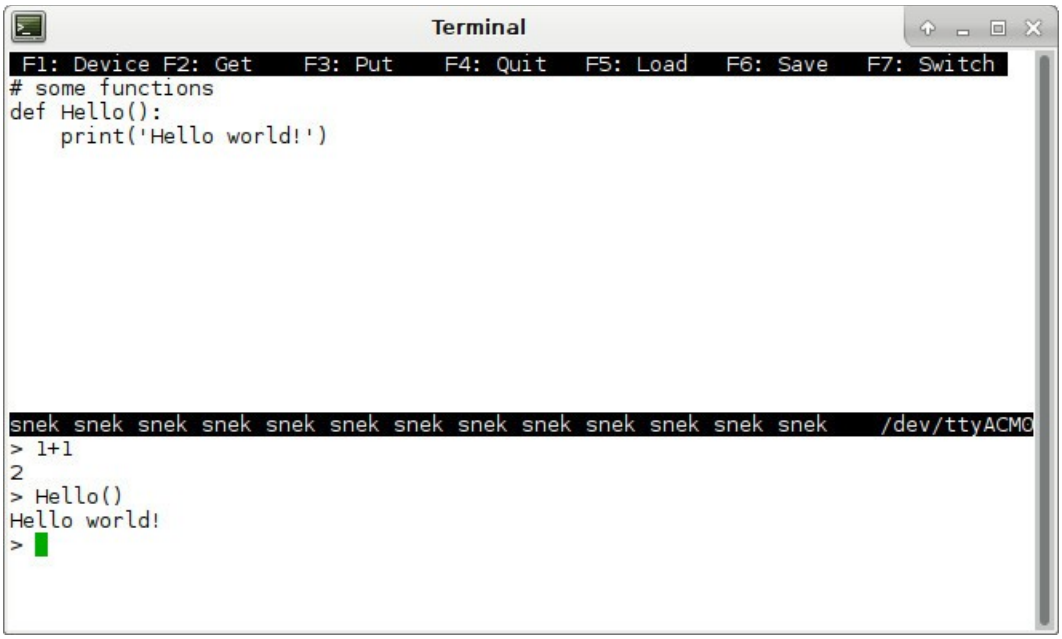
The snek interface makes use of the the laptop function keys **`f1`** through **`f7`** (at the top of the keyboard) that allow you to:

- **`f1`**: Select a device for the interface to connect to through the USB cable. The cable has to be plugged into both the laptop and the snekboard, and the snekboard must be on for this to work. You have to do this before **`f3`**, **`f4`**, or the interpreter work. Use the up and down arrows to move the selection, then enter to choose.
- **`f2`**: Get the program currently stored in the snekboard and display it in the program editor window.

- **`f3`**: Put the program in the program editor into the snekboard memory and restart the snekboard (run the program).
- **`f4`**: Quit the snek interface. Closing the window also exits the interface.
- **`f5`**: Load a file from the laptop into the program editor.
- **`f6`**: Save the contents of the program editor into a file on the laptop.
- **`f7`**: Switch the cursor between program editor and interpreter windows.

A **typical sequence** might consist of:

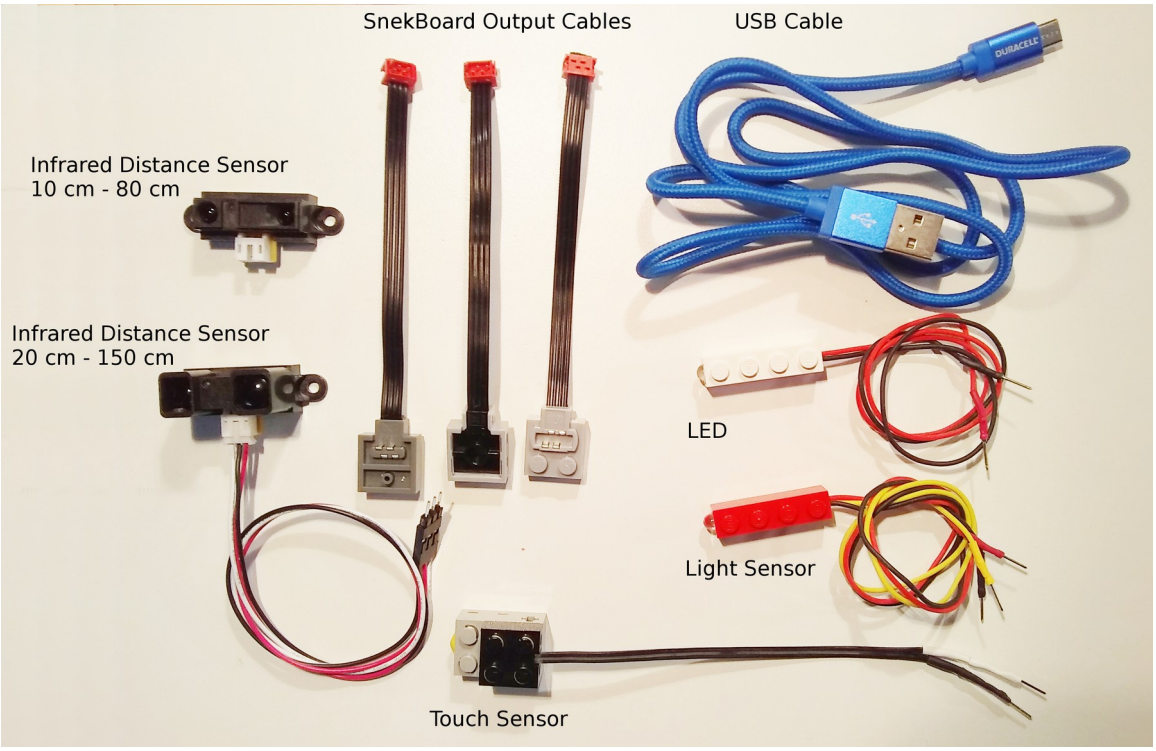
- **`f1`** to connect to a snekboard device,
- **`f2`** to get the program on the board,
- modifying the program (if only to stop automatic execution),
- **`f3`** to put the program on the board,
- **`f7`** to switch to the interpreter window,
- and then entering a command to execute (function call Hello())



Notice that **using snek** will consist primarily of:

- **testing** expressions in the interpreter,
- **defining** functions (both in the interpreter and program editor),
- **figuring out** how to get expressions and functions to work as desired.

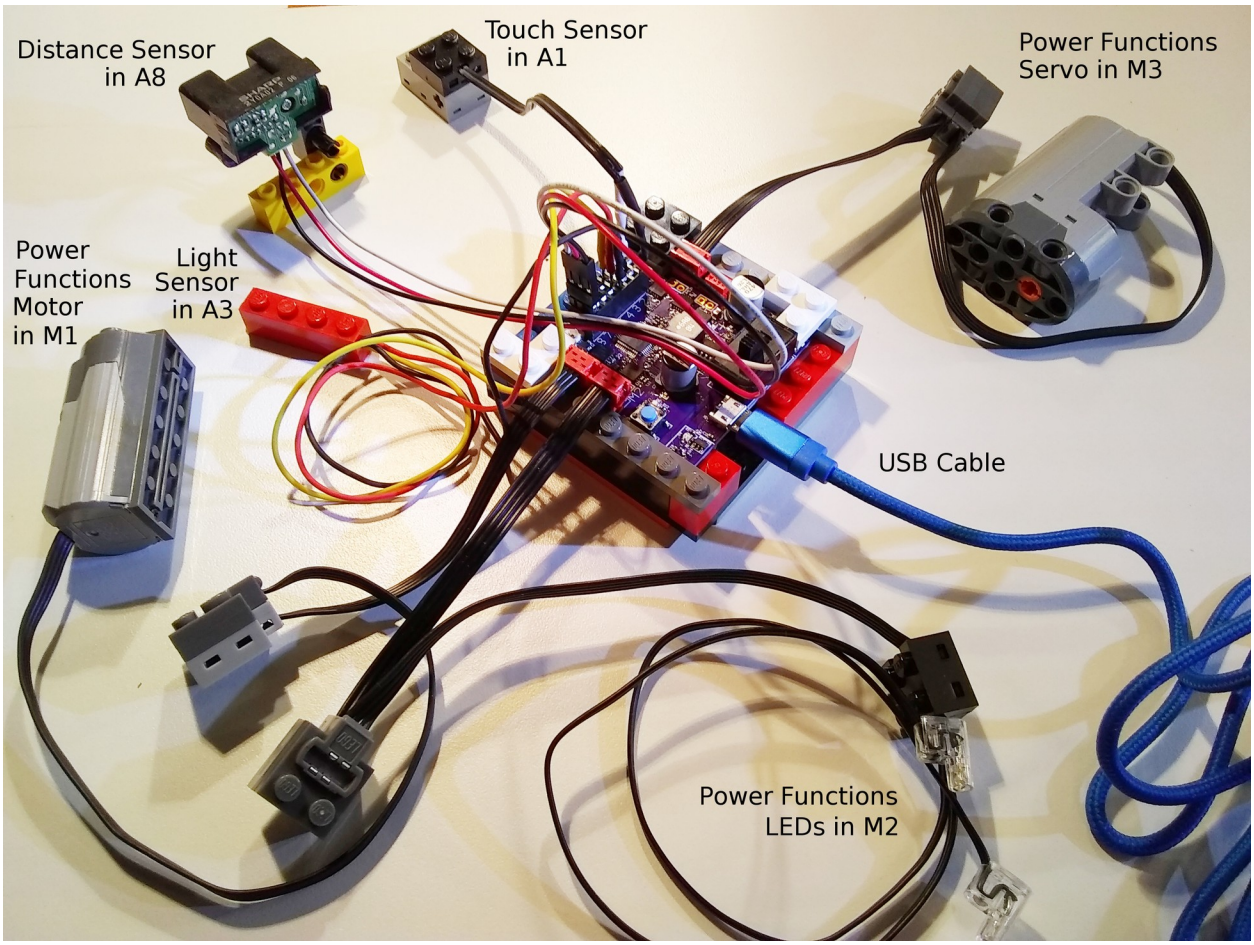
SnekBoard Cables and Sensors



Notice that:

- There are two kinds of **snek output cables**. Light gray connectors have power functions on top and brick underneath. Dark gray connectors have power functions on top and bottom. ***Connect the red connectors to the M1..M4 output ports so that the cables immediately exit the board.***
- The **light sensors** are in red 1x4 Bricks. The white 1x4 brick is an LED, not a sensor, and has no white or yellow sensor signal wire.
- There are two kinds of **distance sensors**. The long range sensor (150cm or about 5 feet) is larger, and the short range one is smaller. The cables are the same for both.
- The **touch sensor** needs a special cable that has a brick connector on one end and (at least) two pins on the other. If it has only two pins, you need to pull up or pull down the signal line in software (depending on how it is connected) for proper function.
- The sensors have either 2 or 3 **colored wires**. Be careful to connect sensor pins to an analog input column ***so the colors match*** the labels of the photo on page 1.

Here is a well connected snekboard:



Snek Outputs

- Are used to **drive** or **control** actuators. **Actuators** are devices that physically act in the world, like motors, servos, lights, and speakers.
- Use ports **M1** to **M4** and **neopixel**.
- Use these **functions to control output ports M1** to **M4**:
  - **talkto**(M#): Remembers output port # for other output commands.
  - **setpower**(P): Sets the output power to P with 0 <= P <= 1.
  - **setleft**(): Sets the output power to flow left-wise through port #.
  - **setright**(): Sets the output power to flow right-wise through port #.
  - **on**(): Turns the power to port # on.
  - **onfor**(S): Turns the power to port # on for S seconds (approximately).
  - **off**(): Turns the power to port # off.
- The **neopixel** LEDs work a bit differently. See the snek manual[a].

a For more complete information see *The Snek Programming Language* by Keith Packard. We currently have version 1.1. installed, but that may change even if these instructions do not. You can find the latest version of Keith's snek including *The Snek Programing Language* at keithp.com. Thanks for getting us started Keith!

Snek Inputs

- Are used to **read** sensors. **Sensors** are devices sense or measure physical conditions in the world such as light intensity, distance (as reflected by IR light), and touch.
- Use **analog input** ports **A1** to **A8** to provide values that range from 0 to 1. “Analog” indicates that the values provided represent continuous values.
- Use the **read function** to get values from ports **A1** to **A8**:
  - **read**(A#): Gives the value (between 0 and 1) of port #.
  - **Advanced Topic**: A 2-wire touch sensor needs pullup(A#) or pulldown(A#). More on this later. A 3-wire touch sensor does not.

Snek Control Flow

Dig into the snek manual to figure these **control statements** out. It has a nice tutorial and complete descriptions about using these. Notice the use of **colons** to signal the end of a test expression and subsequent indentation of the corresponding **code block**. Code blocks must be intented.

- **if**: Conditionally executes code as decided by the **condition** (value of the expression between the if and colon). The **if block** (1st indented block) is executed when the condition is **true** (non-zero). Otherwise the **else block** (2nd indented block) is executed. This allows code to decide what to do.

```
if read(A3) > 0.5:
     on()
else:
     off()
```

  The `else:` and 2nd block are optional. The 1st block is required, but see the pass statement below.
- Snek has two looping control flow statements:
  - **for**: Executes or **iterates** the **loop block** (subsequent indented lines) the number of times specified by the expression between in and colon:

```
for i in range(0,10,1):
     print i
```

  - **while**: Iterates the loop block as long as the **condition** (the value of the expression between while and colon) is **true** (non-zero):

```
while not read(A1):
     onfor(0.5)
```

- **pass**: When you need to test a condition, but not perform any action as the block of a control statement, use the `pass` statement to do nothing except fill in the control statement body. Notice that `pass` is a statement, not a function call – don't use parentheses after it.
- **Function calls:** For example, `Hello()`, `on()`, or `read(A3)` execute the function using the parenthesized list of parameters. Note that the parenthesized list of parameters must be given for a function call even when the function has no parameters.
- **def**: Starts the definition of a function. Functions can also return values as shown in the line bug project. The first line of the definition gives the function name and the list of parameters (data it needs) in parentheses:

```
# TurnUntilSensed assumes already talking to turn motor!
def TurnUntilSensed(sensor,turnSecs):
     while not read(sensor):
          onfor(turnSecs)
```

Coding Tips

- Use the interpreter window to **test soon and often**.
- Use **names** that help you remember the meanings of code and variables. You are defining your own language to talk to the computer! Do it wisely.
- When you need to give more information than specified by the code, use **comments** (lines beginning with the pound sign **#**) to embed the information in the code itself so that someone else (including your later self) can understand what you are doing.

Hopefully this is enough to get you started with snek and the snekboard. You'll soon need to dig into the snek manual for more detail. This is the way of computer science. All of the words, phrases, grammar, and punctuation are very precise in both use and meaning, and you continually need to test and retest your understanding of what you are saying.

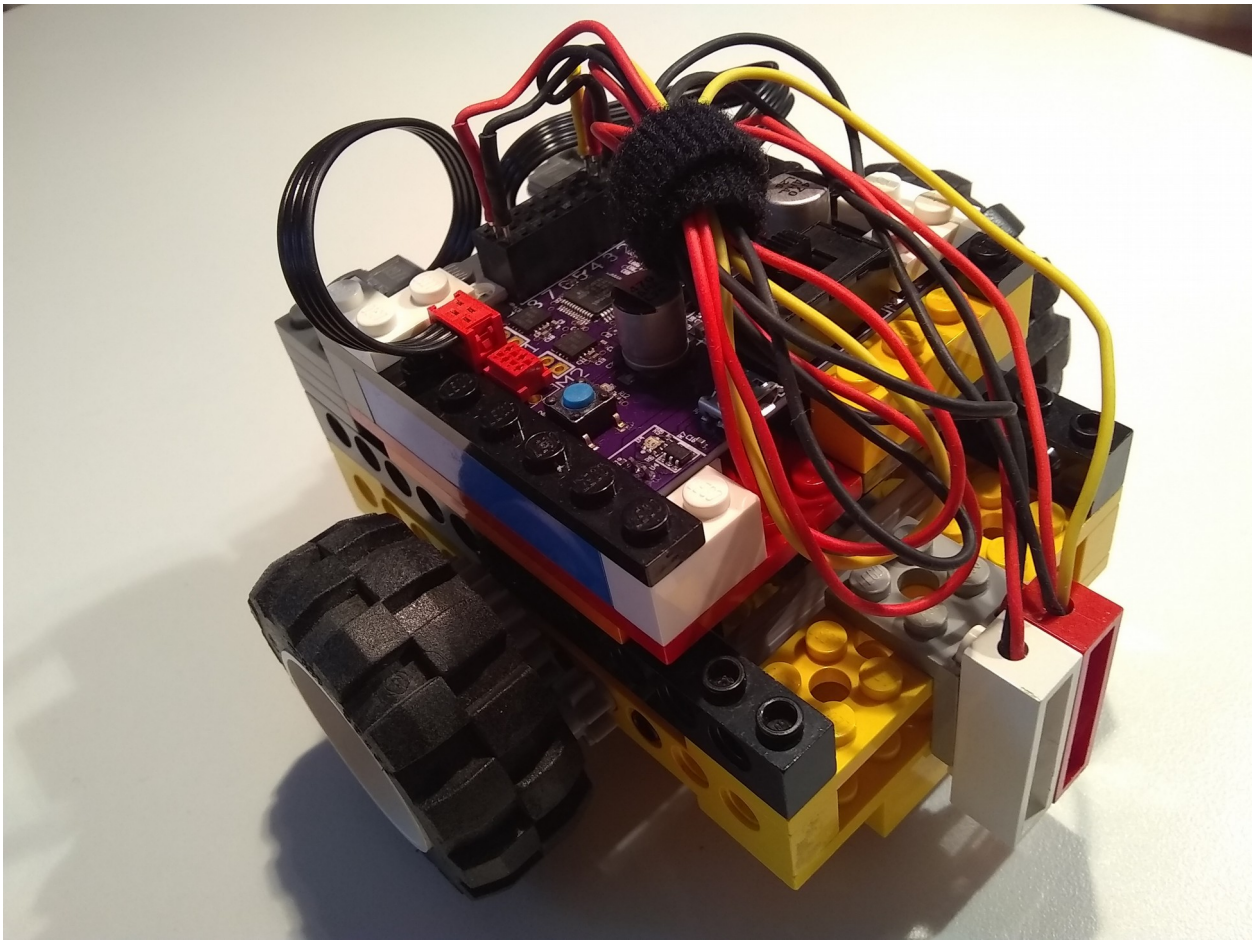
Fill in Your Lego Logo Program Translation Sheet (Snek Part)

One seemingly enjoyable way to do this is to arrange the cursor to be in the interpreter window, and then:

- Work one page at a time.
- Try commands by typing them in and executing them.
- Figure out what the commands are doing by trying variations.
- Write up the answers to one page before proceeding to the next.
- **TEST YOUR ANSWERS!**

The Line Bug Project

Once you have filled in the snek part of your Logo program translation sheet, you are ready to start the line bug project. In this project you will program (and possibly build) a **line bug**: a little mobile robot (bug) to follow a line. There may already be a bug built and ready for you to program, but if not, they are not hard to construct. Instructions below show you how to build one that looks like this:



Important features:

- **Two motors** to move the bug, one for each wheel.
- **One light level sensor**: the red 1x4 brick.
- **One LED** (always on) to illuminate the line so it can be sensed.
- **Skid plates** on the bottom mean the surface must be flat!

There are various **strategies** for solving the challenge of getting the line bug to follow the line (some of which you can pursue all the way to advanced control projects through independent study). However all of them involve sensing the line (a good place to start) and controlling the wheels to keep the bug (and more importantly, the sensors) following the line:

- How will you detect the line? (**Try it!**)
- How will you move the bug based on this? (Start **small** and keep **testing**!)

Sensing the Line

To start you'll need a surface with a line, a light sensor, and some code to read and print light sensor values. You might use a white surface with a black line, or a black surface with a white line. The line needs to be thick enough to detect the line as the bug moves: ¾ of an inch or more. Whatever your intended track, begin by testing the light sensor using something like this (use `<Ctrl>C` to stop):

```
while 1:
    print(read(A1))
    time.sleep(1)
```

Consider:

- **Sensor values will depend on light** reflected from the surfaces (which depends on how much light there is in the area). To make your program less sensitive to this, **shine consistent light with the LED**.
- **Sensor values will vary**. Read a number of different values for the line and use a typical or average value. Do the same for the background. Use these to figure out a threshold half way between the two. [Extensions: Use a well named variable for the threshold, use snek to compute average values and threshold, automate your bug to learn these values …]
- Using **functions and variables to organize and document** the code. One strategy is to name the condition and then return `True` or `False` to indicate the condition, such as OnLine() or OverTape().

For example, if we collect the following values, then typical values are 0.9 and 0.3, and a reasonable threshold would be about halfway between: 0.6. Code to detect the difference could be as simple as this:

White Board	Black Tape
0.9106227	0.3006105
0.9103785	0.3094017
0.9020757	0.3103785
0.9108669	0.3098901
0.8984127	0.3064713

```
if read(A1) > 0.6:
        print("board")
else:
        print("tape")
```

On the other hand, for code that others (including your future self) can more easily understand, test, and tweak, create names for values at the beginning of the program:

```
LightSensor = A1
TapeThreshold = 0.8
```

then, define a well-named function to use them:

```
def OnTape():
     return(read(LightSensor) < TapeThreshold)
```

and use it in later code like this:

```
if not OnTape():
     print("board")
else:
     print("tape")
```

or more appropriately (simpler) like this:

```
if OnTape():
     print("tape")
else:
     print("board")
```

Moving The Bug

How can we keep the bug moving along the line? What if the line is really wide? Let's take this to the extreme: half a rectangle that's white and the other half black. This makes it clear that what is really needed is to **follow the edge** between the two.

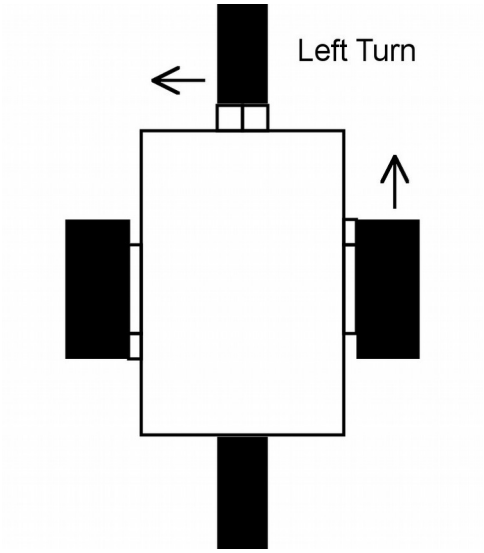
So, how to follow the edge? This is easier if the **edge is near the sensor** (and we allow the bug to turn completely around). Just **turn the bug until it finds the line**. Then what? This is where you either try it and see or I just tell you. For those of you that want to figure it out yourself, when you find the line just stop the bug and take a look and think about it. Or brainstorm with someone. Your test code might look something like this:

```
talkto(M1)
setright()
setpower(1)
on()
while not OnTape():
     pass
off()
```

If you want to proceed this way, I suggest that you make it easier to play (er, .. ah, .. I mean experiment) by organizing with variables and functions as in the last section. Use two functions, one to turn the bug left, the other to turn it right, and get them to work so that each turn moves the bug toward the sensor:

```
RightWheel = M1
LeftWheel = M3
Speed = 1
```

```
def StartLeftTurn():
     talkto(LeftWheel)
     off()
     talkto(RightWheel)
     setright()
     setpower(Speed)
     on()

def StartRightTurn():
     talkto(RightWheel)
     off()
     talkto(LeftWheel)
     setleft()
     setpower(Speed)
     on()
```

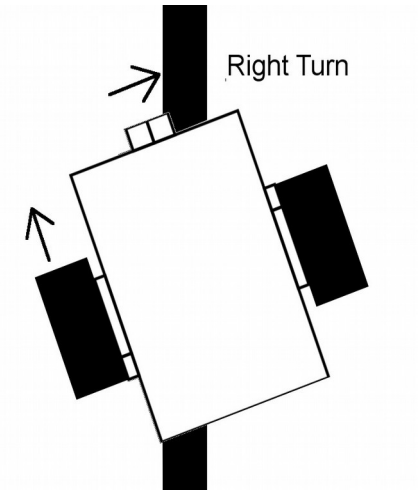


To keep things simple, turn only one wheel at a time. Also notice the exceedingly specific function names. They remind us that they only start a turn and do not complete one. They leave the bug turning. After you get each function working (turn the car upside down to test), control the bug with them at the command line. To stop a turn, just type `off()`. Experiment with a sequence like this that uses the sensor to both start the turn, and complete it:

```
while OnTape():
     StartLeftTurn()
else:
     off()
```

Give them a try. As long as both turns move the bug in the sensor direction, eventually you'll come across the working strategy of **alternating the turns AND alternating the condition**:

```
while OnTape():
     StartLeftTurn()
else:
     off()
while not OnTape():
     StartRightTurn()
else:
     off()
```



The first stage solution can be had by doing this over and over again (and since the turn functions turn off the other motor we leave that part out):

```
def LineBug():
     while True:
          while OnTape():
               StartLeftTurn()

          while not OnTape():
               StartRightTurn()
```

Once you have defined this function, you can test it at the command line by entering: `LineBug()`. Use `<Ctrl>C` to break out of the loop. When you are ready to test it without the USB connection, add the call to `LineBug()` as the last line of the program, with a blank line above it and at the leftmost column. Then pick up the line bug (so it does not run off the edge of the desk and crash apart) and Put (**f3**) the program to write it to the snekboard. If all goes well, the program will start rotating a wheel, disconnect the USB cable, and test the it on your course.

Testing and Debugging

In addition to making code more understandable, using functions and variables (as demonstrated in the last couple of sections) helps us test, tune, and debug[a] our code as well. One of the biggest advantages is that we end up testing **smaller segments of code** that have less to do and make it easier to find where and what is going unexpectedly. Also, by using variables, we can **tweak things without having to change the code** we just got working! You can just change `Speed` and then rerun `LineBug()` to see what happens.

Using Well Named Variables and Functions:

- Makes the code more **understandable**.
- Makes the code **easier to test and debug**.
- Makes it **easier to try** ideas.
- Builds a **language for communicating** your ideas.

Nevertheless, even using these strategies, things often still go wrong. The first step of debugging is to think about what the code is doing in detail (see *Optimizing Code* on page 16). If that doesn't do the trick, print things that tell you what the code is doing (so you can check it against what you think it should be doing)! The basic idea is to print out what the code is doing at strategic points (often printing out key values as well):

a Debugging means making the code work as intended. Originally it was about keeping actual bugs (like flies) out of the mechanisms (like relays) of computing machinery.

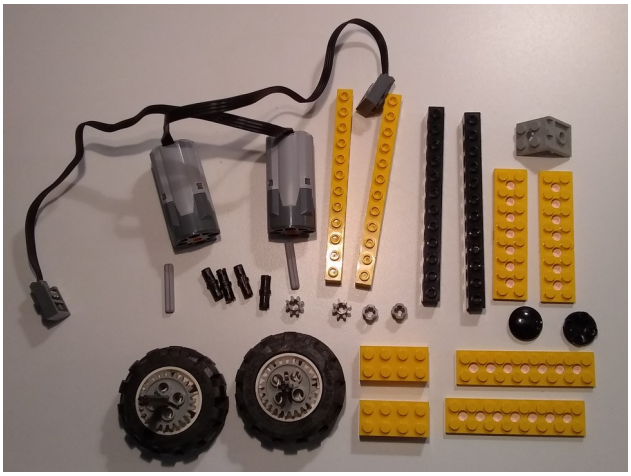
```
def LineBug():
     while True:
          print(“Turn left while on tape.”)
          while OnTape():
               StartLeftTurn()

          print(“Turn right while off tape.”)
          while not OnTape():
               StartRightTurn()
```

Until you start writing code based on ideas not easily visible from the code itself, you are better off using meaningful variable and function names than writing lots of comments. Since comments are not executed, tested, and subsequently corrected, they often are wrong. Nevertheless, we’ll use a few comments (once you get your line bug working) and keep them around for both description and future debugging:

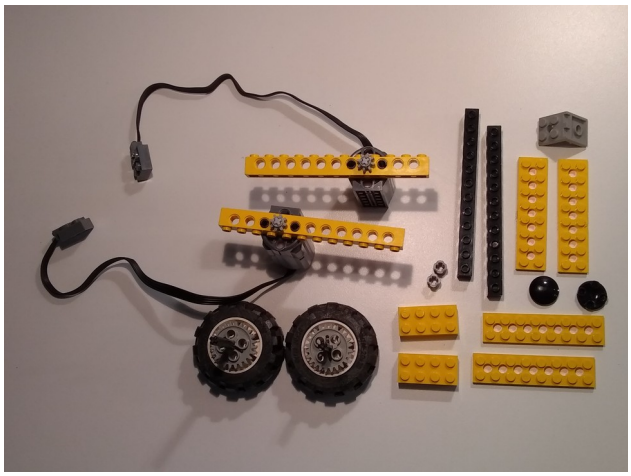
```
def LineBug():
     while True:
          #print(“Turn left while on tape.”)
          while OnTape():
               StartLeftTurn()

          #print(“Turn right while off tape.”)
          while not OnTape():
               StartRightTurn()
```

Building The Line Bug

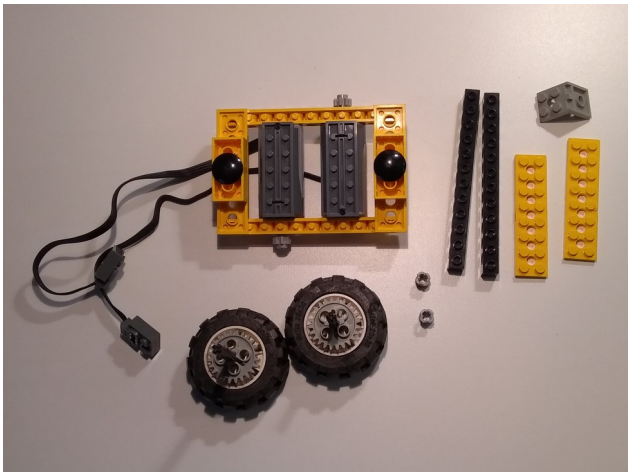
Step 1: Gather the parts.



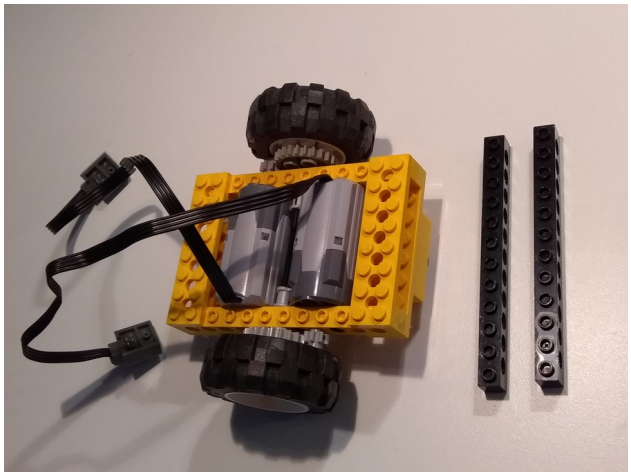
Step 2: Attach the motors.



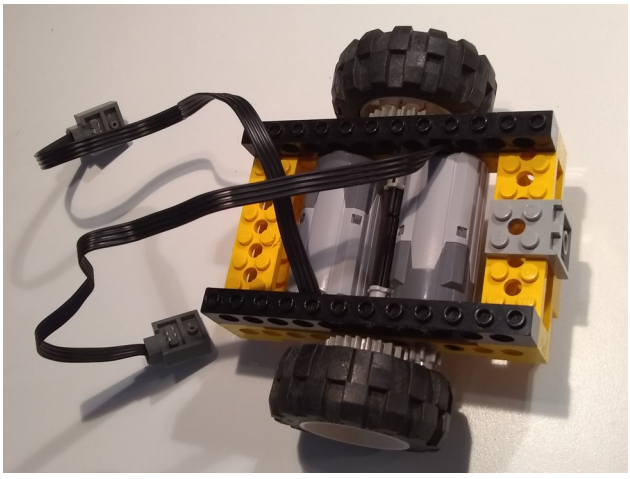
Step 3: Attach the plates and skids.



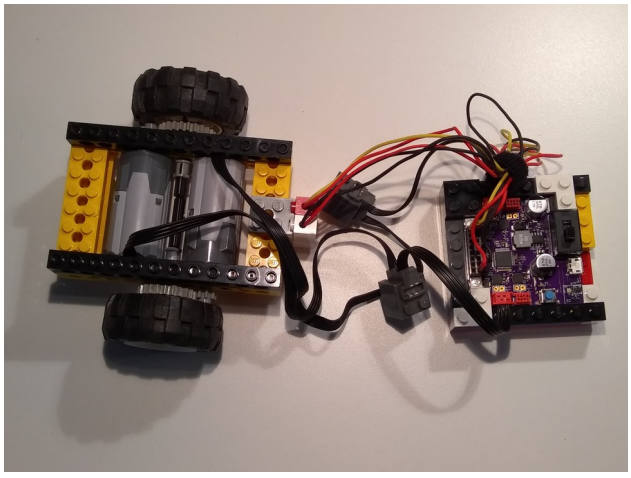
Step 4: Attach the plates and wheels.



Step 5: Attach the riser beams.



Step 6: Attach the bricks and cables.



Step 7: Attach the snekboard. See the photo on page 8.

Advanced Topics

Optimizing Code

Once you get a simple piece of code working, you often want it to do more and/or do it faster. Here, we'll take a look at the common strategy of **doing only what needs to be done** to make software more efficient. One of the best places to look for the **most impactful improvements** is in code that gets executed most repeatedly: **loops** (within loops, within loops …). For example, you may find that you like your line bug to be quicker about detecting and responding to the edge of the line. The place to look is one of the loops:

```
while OnTape():
     StartLeftTurn()
```

Think about everything the code is doing. Talk it through with a partner if you can. Go through a number of loops including the termination. Here we have two loop iterations before exiting the loop:

```
. . . (turning left looking for edge: right motor is turning)
read the light level sensor and compare to the threshold
the value is less than the threshold so
     stop the left motor
     start the right motor turning right at power
read the light level sensor and compare to the threshold
the value is less than the threshold so
     stop the left motor
     start the right motor turning right at power
read the light level sensor and compare to the threshold
the value is greater than the threshold so skip over the loop
```

I've highlighted the actions that we don't need (left motor is already off, right already turning). As a first step, we can restructure the code to not execute these unnecessary actions and only use the condition test part of the loop:

```
def LineBug():
     while True:
          StartLeftTurn()
          while OnTape():
               pass
```

```
StartRightTurn()
while not OnTape():
    pass
```

This involves taking no action within the loop. However, python and snek require a loop body, so we use the pass statement. The actions for the same situation would now look like this:

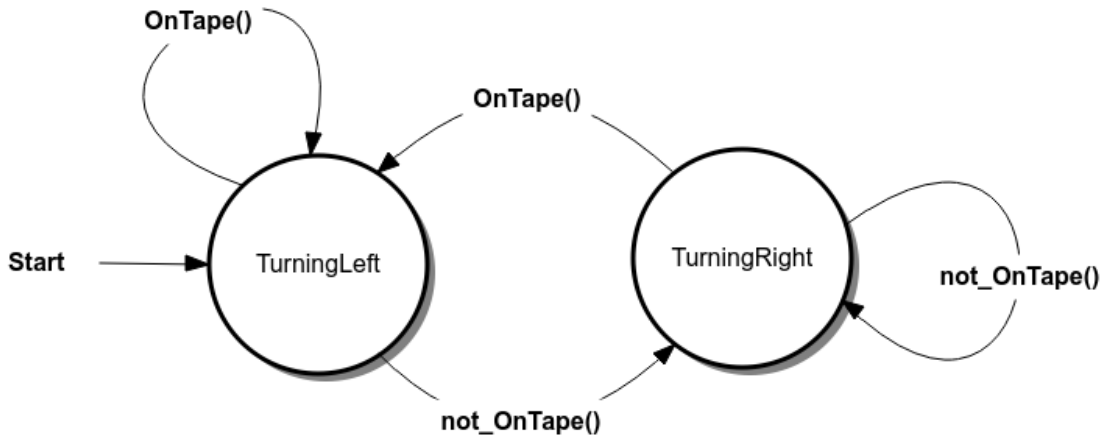
. . . (turning left looking for edge: right motor is turning)
read the light level sensor and compare to the threshold
the value is less than the threshold
read the light level sensor and compare to the threshold
the value is less than the threshold
read the light level sensor and compare to the threshold
the value is greater than the threshold so skip over the loop
stop the left motor
start the right motor turning right at power
read the light level sensor and compare to the threshold
the value is greater than the threshold
read the light level sensor and compare to the threshold
the value is greater than the threshold
. . . (continue until on tape: less than threshold)

This time the bold actions are for the next turn direction. Examining behavior of code in this way has a number of useful effects:

- It helps you better **understand what the code is doing** (for both debugging and optimization). Write things down and use diagrams.
- It helps you figure out **how to restructure code** to do only what is necessary.
- It helps you **focus on frequent actions** in order to improve efficiency.
- It helps you **build a knowledge base** for understanding, designing, coding, testing, and debugging code.

It is also important to look at how we changed the code at a higher lever. We started with the most straight-forward approach to the line bug with code that performed control actions in a simple, straightforward way. Then, in this section, we focused on efficiency and modified the code to take action only when necessary. This is a step toward **event based software** that is structured to take action in response to events (such as a change in condition via light sensor as above). This is often combined with **state based software** that is structured to have well defined responses to events depending on what state the system is in.

Often this combination is condensed into a crystallized form know as **finite state machines** with diagrams to depict their structure:



You can think of our code above as breaking naturally into these two states, where the **action upon entering a state** is to turn off one turn and start the other. This is a common pattern. Also common is **action when leaving a state**, that often depends on the event that causes the **transition from one state to another**. Even without tools[a] to edit programs this way, you can see how sketching the diagrams and thinking this way can help understand and design software and behavior.

Analog Input Smoothing

One thing you may notice when working with distance sensors is that the readings can vary drastically. The following samples are all for an unchanging distance to a wall (max 0.1320, min 0.1193):

0.1261	0.1251	0.1290	0.1261	0.1251	0.1251	0.1251	0.1261	0.1261	0.1251
0.1251	0.1251	0.1261	0.1261	0.1251	0.1251	0.1320	0.1261	0.1251	0.1251
0.1193	0.1261	0.1261	0.1251	0.1251	0.1300	0.1261	0.1251	0.1251	0.1251
0.1261	0.1261	0.1251	0.1251	0.1251	0.1261	0.1251	0.1251	0.1251	0.1320
0.1261	0.1251	0.1251	0.1193	0.1261	0.1261	0.1251	0.1251	0.1290	0.1251

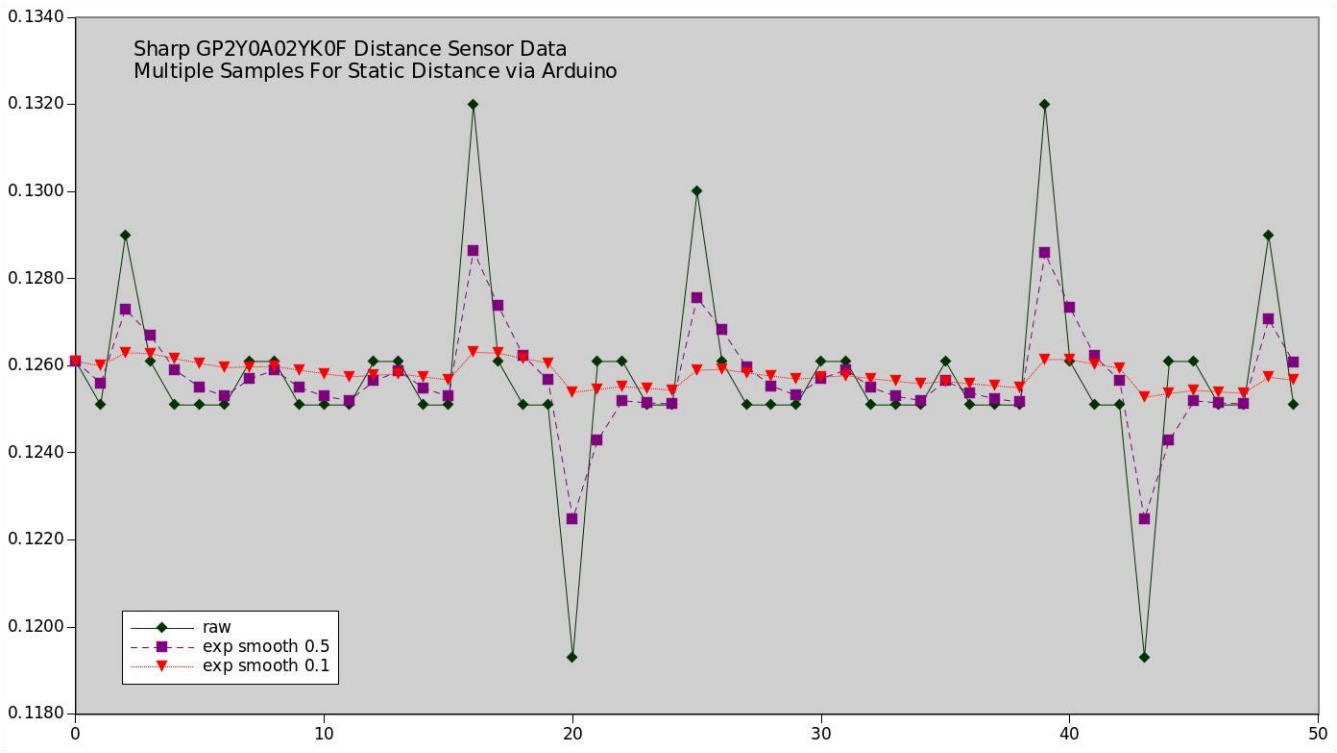
Sometimes, this is too much and the wild values trigger some behavior that you want to do only if you are more certain of the distance. One way to try to work

a In case you use linux, you can play around with the editor I used to generate this: Qfsm. You can find it here: https://sourceforge.net/projects/qfsm/

with noisy data is to smooth it out using an update factor to incorporate the new reading into a smooth value like this:

```
factor = 0.5
smooth = smooth*(1.0-factor) + read(A1)*factor
```

You then use the `smooth` value rather than the new sensor value you just read from `A1`. You can see the effect of some different update factors in this graph:

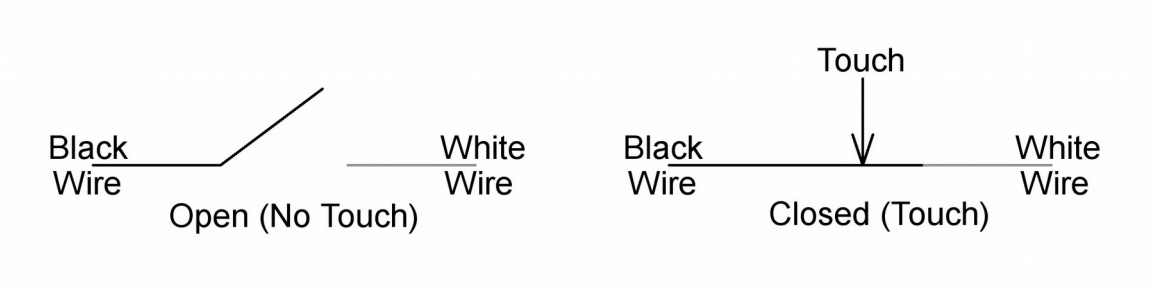


This is frequently called **exponential smoothing** and/or **exponential filtering**[a], because the difference between the actual value and smoothed value over time fits an exponent function exp(kt) for an appropriate choice of constant k. **Beware** that: *the smoother you make the data (the smaller the update factor), the longer (number of samples) it takes for changes in data to show up*!

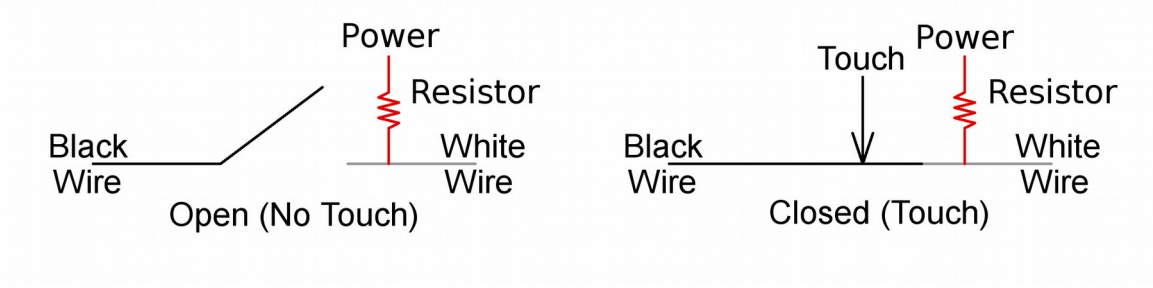
a This explanation seems short and to the point (but don't worry about it if it is not understandable): https://gregstanleyandassociates.com/whitepapers/FaultDiagnosis/Filtering/Exponential-Filter/exponential-filter.htm

2-Wire Touch Sensors and Pull Up/Down

A touch sensor is a switch, open when not touched and closed when touched:



On a 2-wire touch sensor, when the switch is open, the value of the analog input is undetermined (can range from 0 to 1). When the switch is closed, the value of the analog input is the same as the black wire (usually connected to ground at 0 Volts, giving a reading of reading of 0). In this case (black wire connected to ground), we can tell the snekboard to hook the analog input (say `A1`) to the power line internally



via `pullup(A1)` so that `read(A1)` gives the value 1 when the switch is open and the value 0 when the switch is closed. A 3-wire touch sensor has an external connection to power. This gives reverse logic[a]: 0 ↔ touched, 1 ↔ not touched.

Using the water flow analogy[b] for electricity, we can think of **voltage like water pressure**, **current like water flow**, and **wires like pipes**. We can think of the open end of the white wire like a bucket connected by a white pipe to a our water pressure meter corresponding to our `A1` analog input. If it is raining, the bucket collects water and we read a non-zero voltage. The connection to power through the resister is like a trickle keeping the bucket full. When we touch the sensor to close the switch, it connects the bucket to a drain though a big black pipe that empties the bucket much faster than the trickle can fill it.

a Alternatively, attaching the black wire to power instead of ground and pulling the analog input down instead of up gives intuitive logic: 0 ↔ not touched, 1 ↔ touched, but this can damage delicate analog input ports!

b http://hyperphysics.phy-astr.gsu.edu/hbase/electric/watcir.html