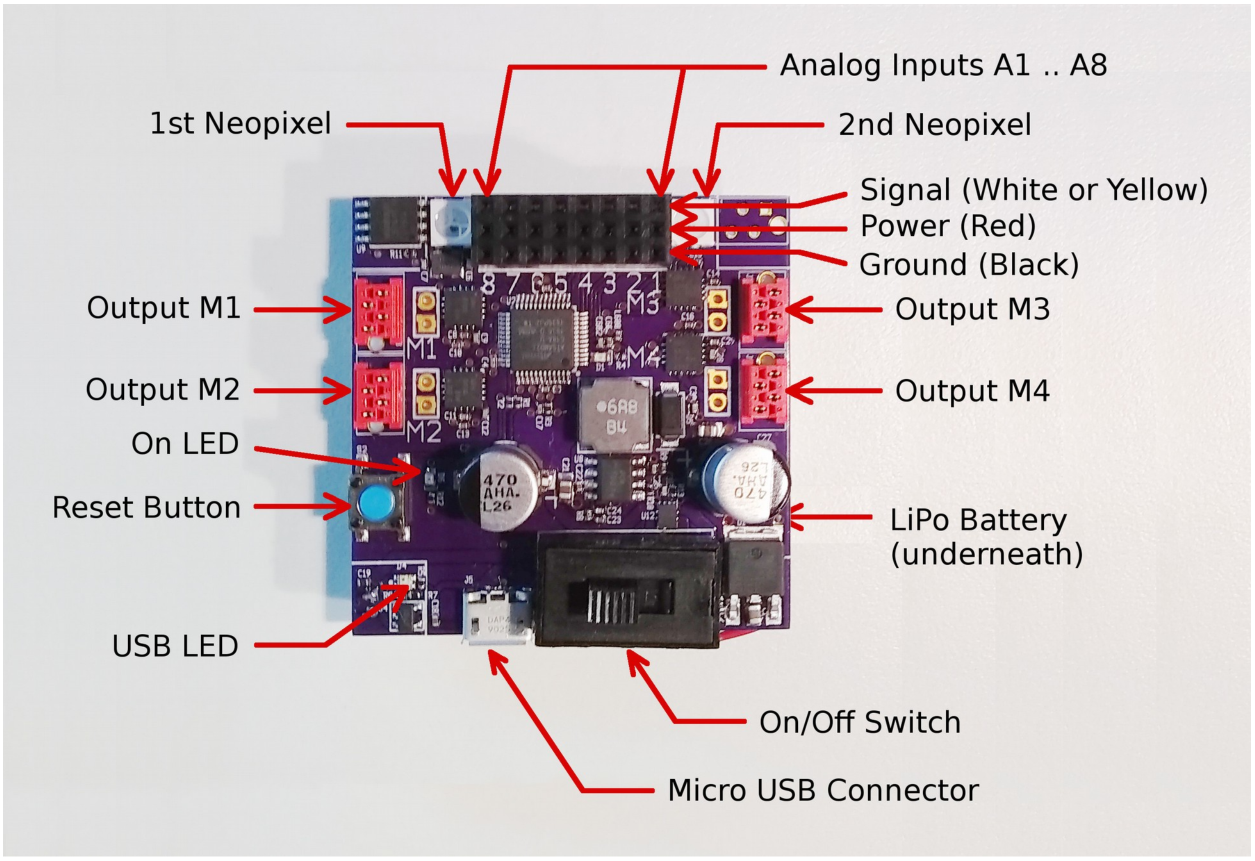
Getting Started With Snek on a Snekboard

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.

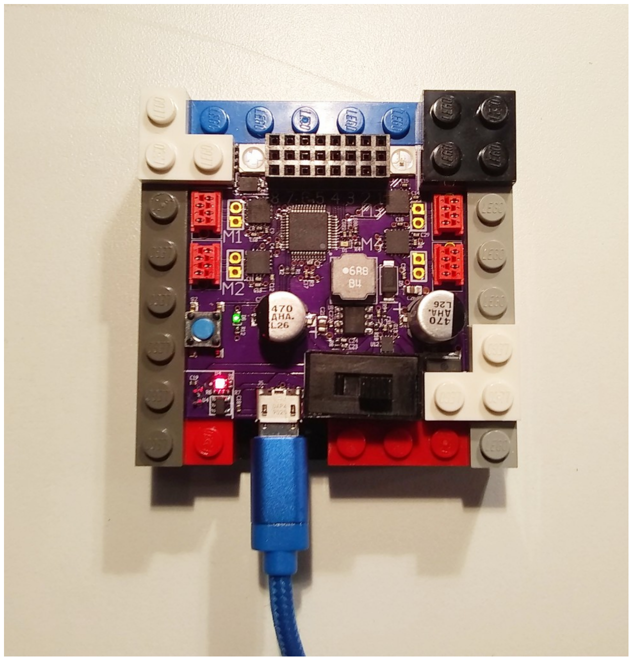
First Look at a SnekBoard

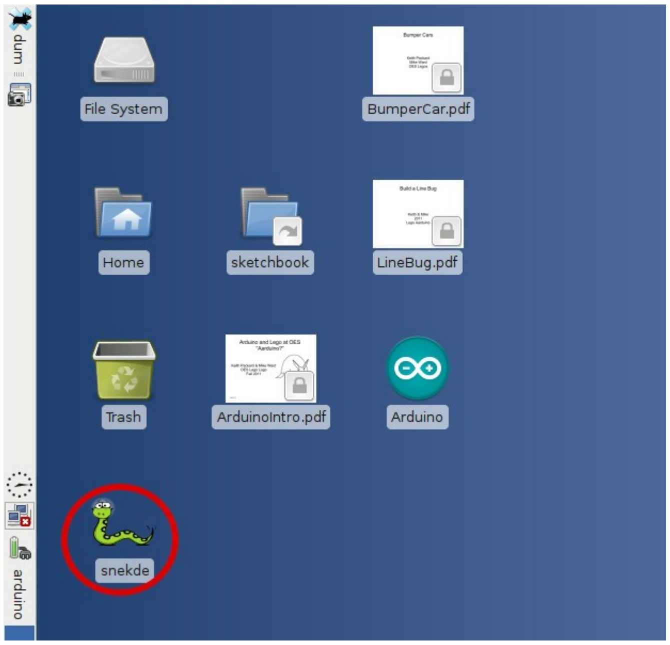


Notice that the snekboard has:

- 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.

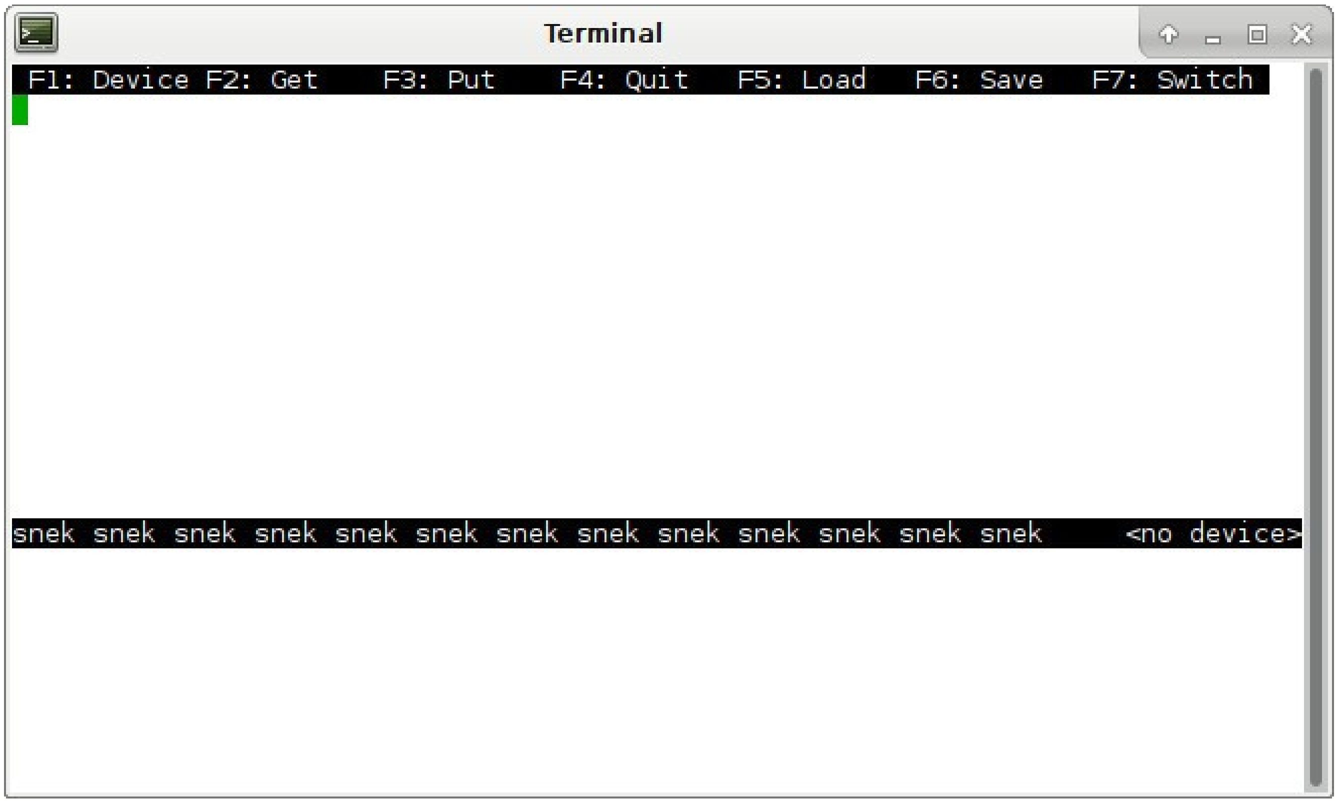
Here is what it looks like when in its Lego case, connected to a laptop, and charging. And here is the Icon for the snek interface (circled in red):





Snek Development Environment (SnekDE)

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



```
 F1: Device F2: Get    F3: Put    F4: Quit    F5: Load    F6: Save    F7: Switch

snek snek snek snek snek snek snek snek snek snek snek snek snek      <no device>
```

Notice that:

- The interface opens with the cursor in the top window. This top window 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**.

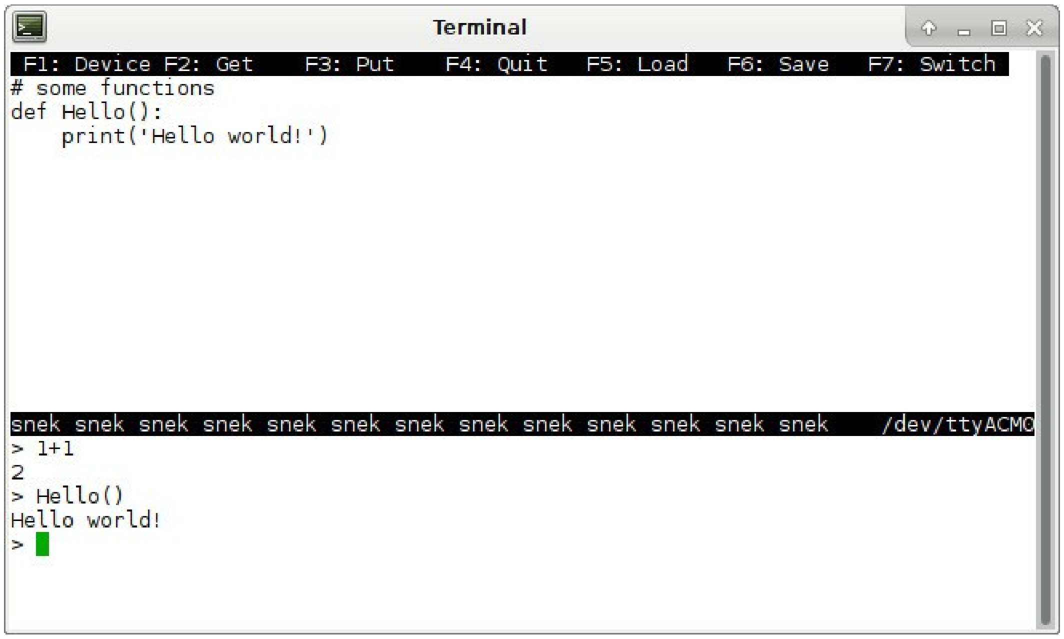
The snek interface makes use of the the laptop function keys **`f1`** through **`f7`** (at the top of the keyboard):

- **`f1`**: Allows you to 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 1st to use the interface. Use the up and down arrows to move the selection and enter to choose.
- **`f2`**: Get the program currently stored in the snekboard and display it in the program editor window.

- **`f3`**: Puts the program in the program editor into the snekboard memory and restarts the snekboard (runs the program).
- **`f4`**: Quits the snek interface. Closing the window also exits the interface.
- **`f5`**: Loads a file from the laptop into the program editor.
- **`f6`**: Saves the contents of the program editor into a file on the laptop.
- **`f7`**: Switches 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())

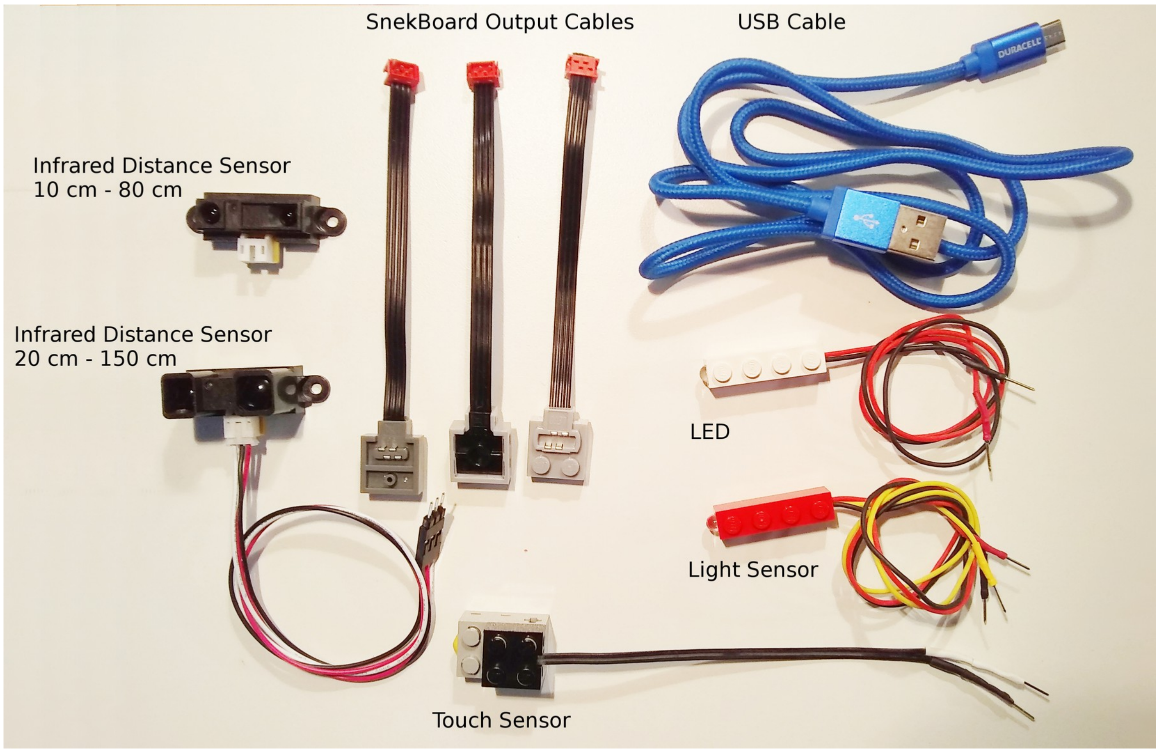


```
 F1: Device F2: Get     F3: Put     F4: Quit    F5: Load    F6: Save    F7: Switch
# some functions
def Hello():
    print('Hello world!')









snek snek snek snek snek snek snek snek snek snek snek snek snek     /dev/ttyACM0
> 1+1
2
> Hello()
Hello world!
> 
```

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.

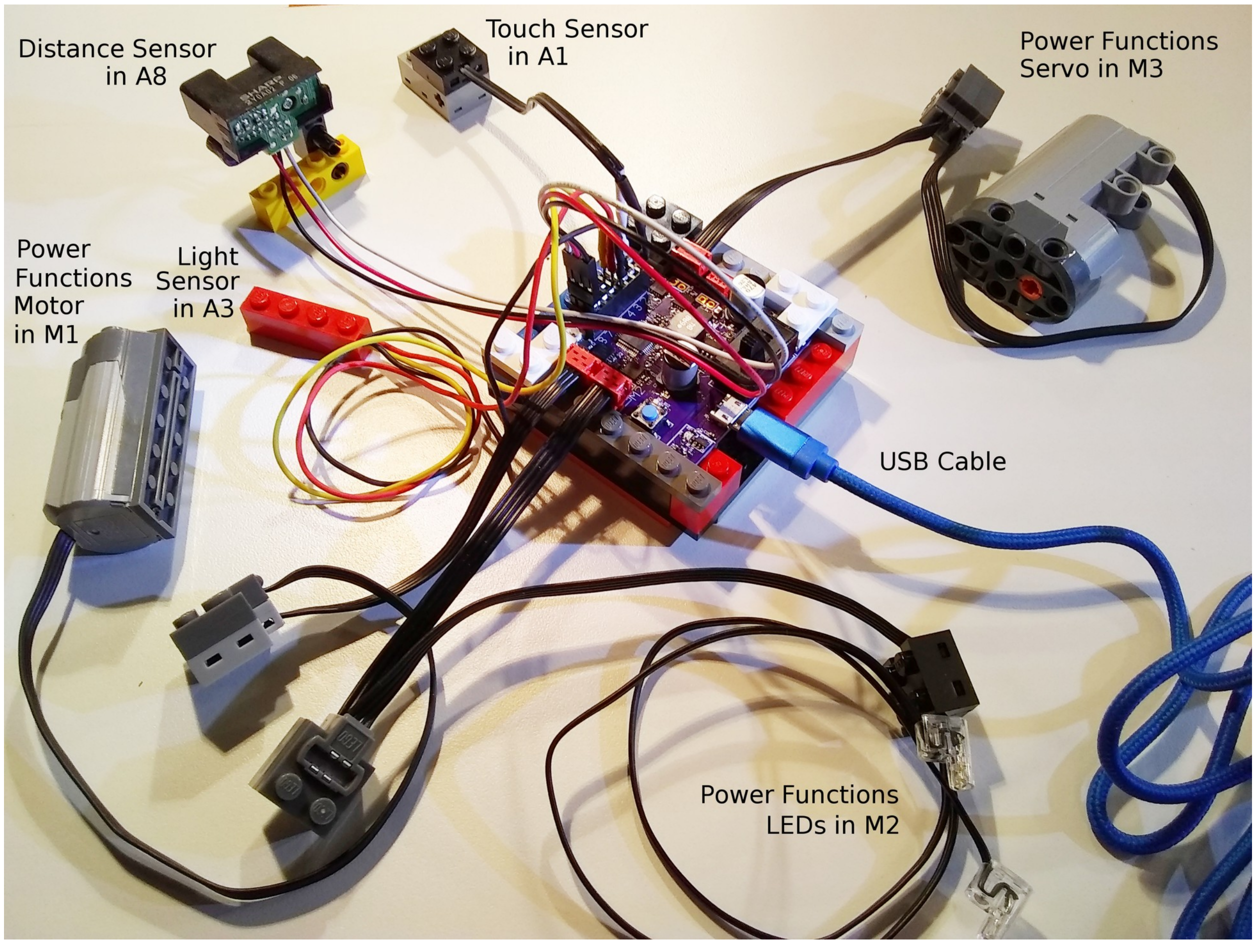
SnekBoard Cables and Sensors



Notice that:

- There are two kinds of **snek output cables**. The light gray connectors have power functions on top and brick underneath. The dark gray connectors have power functions on top and bottom. ***Connect the red connectors 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 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 the 1st page.

Here is a well connected snekboard:


Distance Sensor in A8
Touch Sensor in A1
Power Functions Servo in M3
Power Functions Motor in M1
Light Sensor in A3
USB Cable
Power Functions LEDs in M2

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 the ports **M1** to **M4** and **neopixel**.
- To use output ports **M1** to **M4**, call the following **control functions** to control outputs:
  - **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** actuators work a bit differently. See the snek manual.

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.
- The **analog input** ports **A1** to **A8** provide values that range from 0 to 1. Analog indicates that the values provided represent continuous values.
- To get analog inputs from ports **A1** to **A8**, call the read function:
  - **read**(A#): Gives the value (between 0 and 1) of port #.
  - **Special topic**: A 2-wire sensor needs pullup(A#) or pulldown(A#). More on this later. A 3-wire sensor does not.

Snek Control Flow

Dig into the manual to figure these 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**.

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

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

  The else: and 2nd block are optional.
- Snek has two looping control flow commands:
  - **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 while condition (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 parenthases after.

- **Function calls:** For example, Hello(), on(), or read(A3) executes the function using the parenthesized list of parameters. Note that the parenthesized list of parameters must be given even if there are no parameters.
- **def**: Starts the definition of a function. Functions can also return values as shown in the linebug 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 Programming Language* manual by Keith Packard[a]. This is the way of computer science. All of the words, phrases, grammar, and punctuation are very precise in meaning and use, 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)

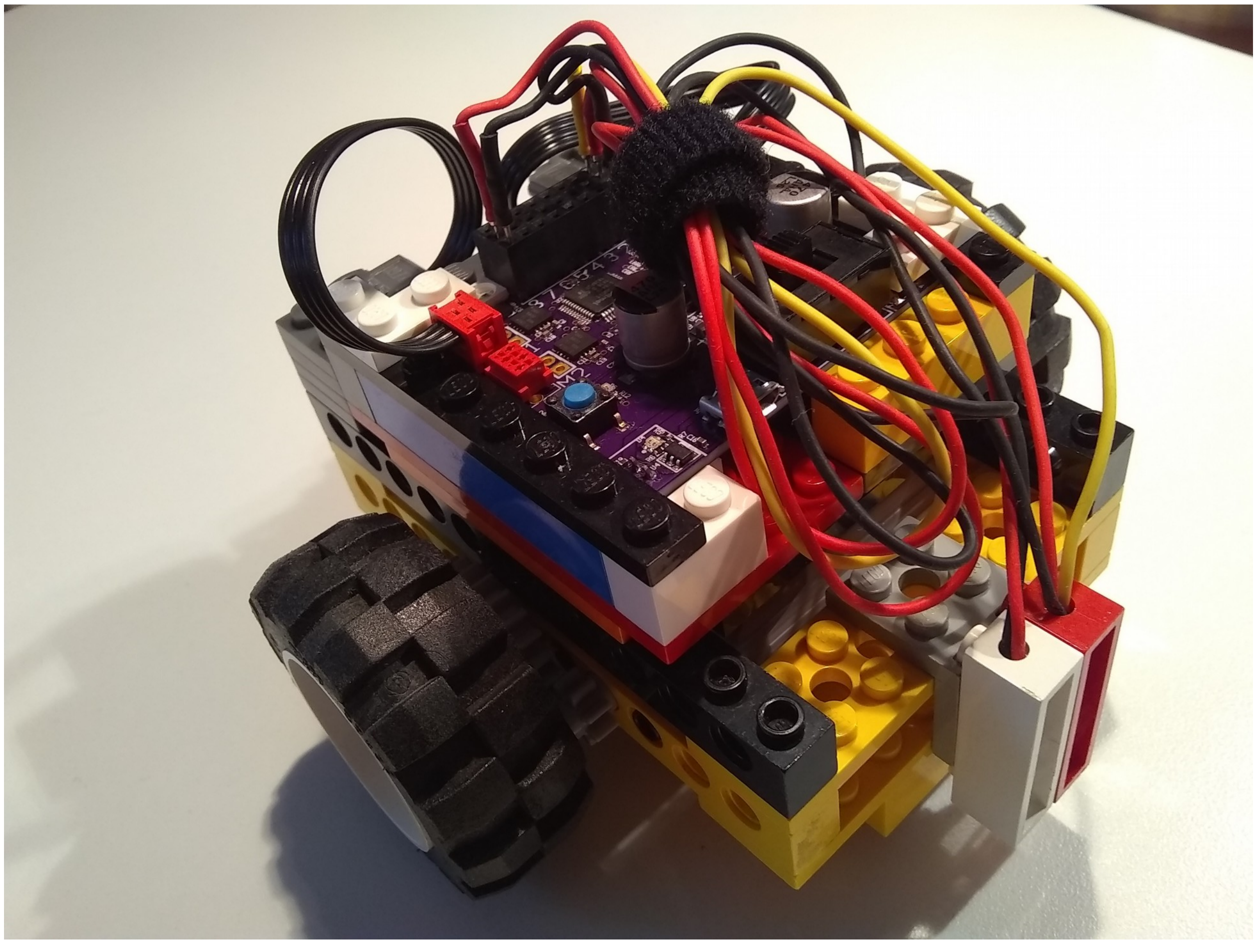
One seemingly enjoyable way to do this is to arrange the cursor to be in the interpreter window, and then:

- 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!

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!

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 (possibly build) and program 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 uniform 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. However, whatever your intended track, begin by testing the light sensor on it 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**.
- The **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 …]
- Use well named **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 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 the well proven strategy to **follow the edge** pop right out.

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. 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:

```
    StartLeftTurn()
    while OnTape():
        pass
    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**:

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

So the first stage solution can be had by doing this over and over again:

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

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

You can test this at the command line too. 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, you can disconnect the USB cable, and test the line bug 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 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. One simple and useful way of debugging is to print things that tell you what your code is doing (so that 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):

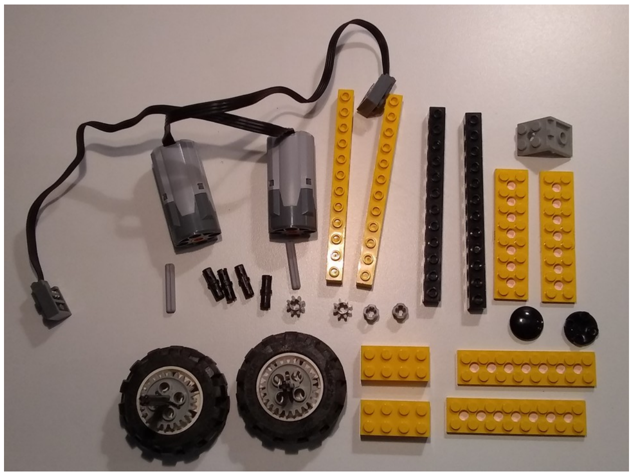
```
def LineBug():
    while True:
        print(“Start left turn and wait for board.”)
        StartLeftTurn()
        while OnTape():
            pass

        print(“Start right turn and wait for tape.”)
        StartRightTurn()
        while not OnTape():
            pass
```

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:

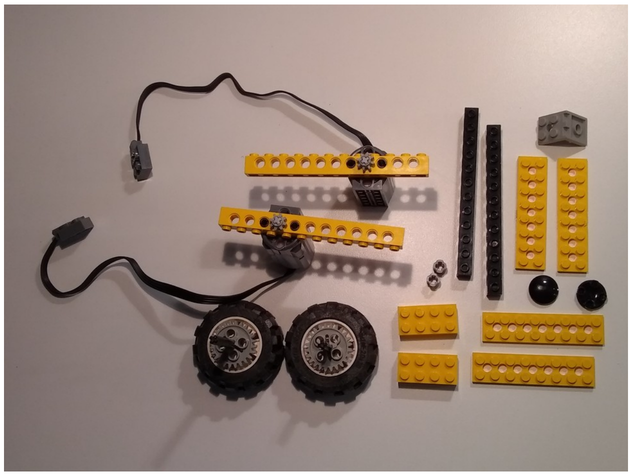
```
def LineBug():
    while True:
        #print(“Start left turn and wait for board.”)
        StartLeftTurn()
        while OnTape():
            pass

        #print(“Start right turn and wait for tape.”)
        StartRightTurn()
        while not OnTape():
            pass
```

Building The Line Bug

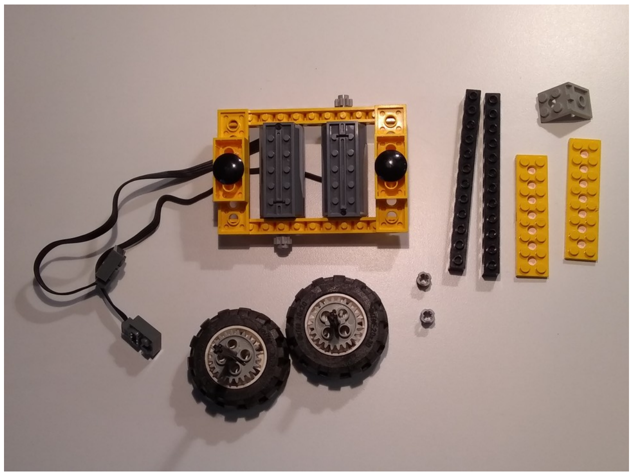
Step 1: Gather the parts.



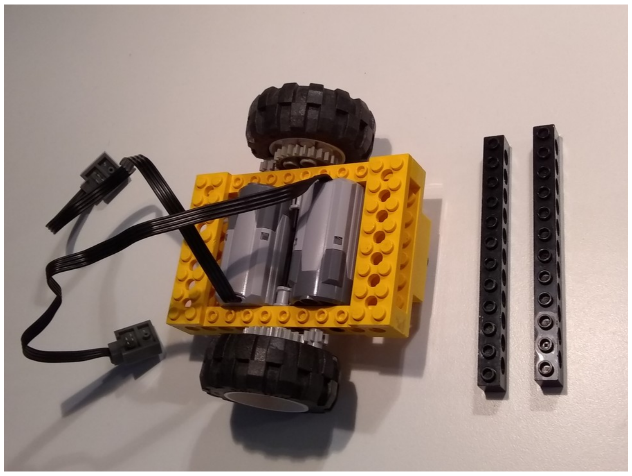
Step 2: Attach the motors.



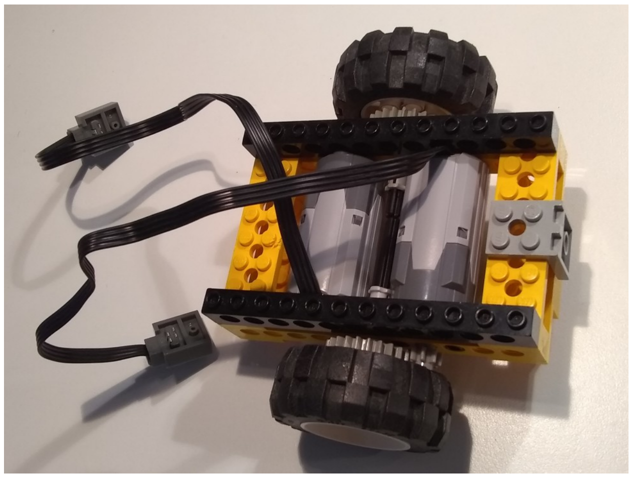
Step 3: Attach the plates and skids.



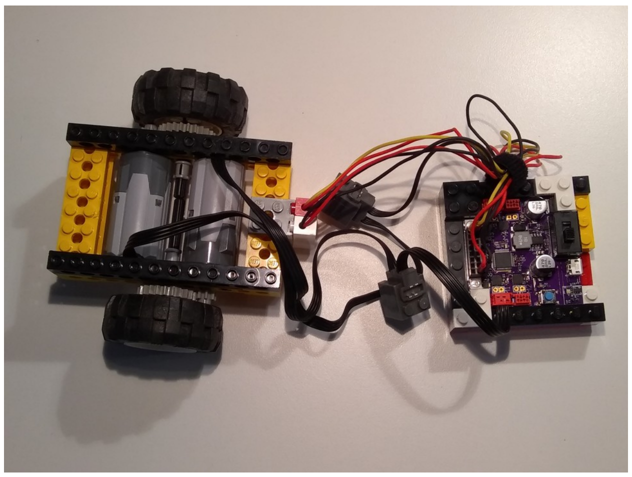
Step 4: Attach the plates and wheels.



Step 5: Attach the riser beams.



Step 6: Attach the bricks and cables.



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