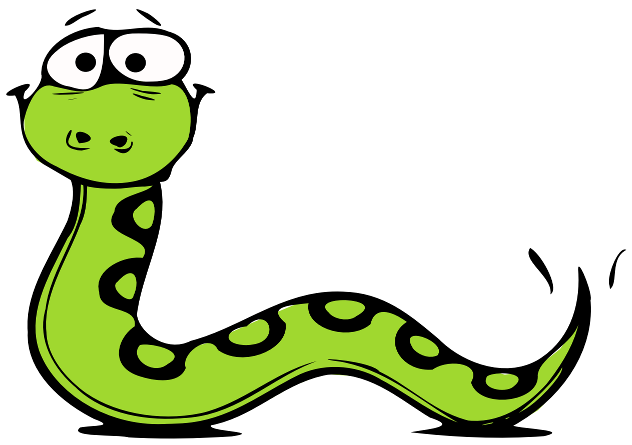

The Snek Programming Language

A Python-inspired Embedded Computing Language

Keith Packard

Version v1.3, 2019-12-15

Table of Contents

License

Acknowledgments

Thanks to Jane Kenney-Norberg for building a science and technology education program using Lego. Jane taught my kids science in elementary school and Lego after school, and let me come and play too. I'm still there helping and teaching, even though my kids are nearly done with their undergraduate degrees.

Thanks to Christopher Reekie and Henry Gillespie who are both students and student-teacher in Jane's program and who have helped teach Arduino programming using Lego robots. Christopher has also been helping design and test Snek.

> Keith Packard
> keithp@keithp.com [mailto:keithp@keithp.com]
> https://keithp.com

Chapter 1. History and Motivations

Teaching computer programming to students in the 10-14 age range offers some interesting challenges. Graphical languages that construct programs from elements dragged with a mouse or touch-pad can be frustratingly slow. Users of these languages don’t develop portable skills necessary for more advanced languages. Sophisticated languages like C, Java and even Python are so large as to overwhelm the novice with rich semantics like “objects” and other higher level programming constructs.

In days long past, beginning programmers were usually presented with microcomputers running very small languages: BASIC, Forth, Logo or the like. These languages were not restricted to aid the student, but because the hosts they ran on were small.

Introductory programming is taught today in a huge range of environments, from embedded systems to cloud-based systems. Many of these are technological dead-ends — closed systems that offer no way even to extract source code, much less to reuse it in another environment.

Some systems, such as Raspberry PI and Arduino, are open — they use standard languages so that skills learned with them are useful elsewhere. While the smallest of these machines are similar in memory and CPU size to those early microcomputers, these smaller machines are programmed as embedded computers using a full C++ compiler running on a separate desktop or laptop system.

1.1. Arduino in the Lego Program

I brought Arduino systems into the classroom about five years ago. The hardware was fabulous and we built a number of fun robots. After a couple of years, I built some custom Arduino hardware for our needs. Our hardware has screw terminals for the inputs and outputs, a battery pack on the back and high-current motor controllers to animate the robots. Because these platforms are Arduino (with an ATmega 328P processor and a FTDI USB to serial converter) we can use the stock Arduino development tools.

Students struggled with the complex syntax of Arduino C: they found it especially hard to type the obscure punctuation marks and to remember to insert semicolons. I often heard comments like “this takes too much typing” and “why is it so picky about semicolons?” The lack of an interactive mode made experimenting a bit slower than on our Logo systems. In spite of the difficulties, there have been students who have done interesting projects in Arduino robotics:

- Chris Reekie, an 11th-grade student-teacher in the program, took the line follower robot design and re-wrote the Arduino firmware to include a PID controller algorithm. The results were spectacular, with the robot capable of smoothly following a line at high speed.

- Henry Gillespie, another 11th-grade student-teacher, created a robot that automatically measured a person’s height. This used an optical sensor to monitor movement of a beam as it lowered onto the person’s head and showed measurements on an attached 7-segment display. We’ve shown this device at numerous local Lego shows.
- Mark Fernandez, an eighth-grade student, built a solar energy system that automatically tracked the sun. Mark is now a mechanical engineering student at Washington University in St Louis.

The hardware was just what we wanted, and a few students used skills learned in the program later on. However, the software was not aimed at young students just starting to write code. Instead of throwing out our existing systems and starting over, I wondered if we couldn’t keep using the same (hand-made) hardware but improve the programming environment.

1.2. A New Language

I searched for a tiny programming language that could run on Arduino and offer an experience more like Lego Logo. I wanted something that students could use as a foundation for further computer education and exploration, something very much like Python.

There is a smaller version of Python, called MicroPython: it is still a fairly large language which takes a few hundred kB of ROM and a significant amount of RAM. The language is also large enough that we couldn’t cover it in any detail in our class time.

I finally decided to just try and write a small Python-inspired language that could fit on our existing Arduino Duemilanove compatible hardware. This machine has:

- 32kB of Flash
- 2kB of RAM
- 1kB of EEPROM
- 1 serial port hooked to a USB/serial converter
- 1 SPI port
- 6 Analog inputs
- 14 Digital input/output pins

I believe that shrinking the language to a small Python subset will let the language run on this hardware while also being simple enough to expose students to the whole language in a small amount of class time.

1.3. Introducing Snek

The goals of the Snek language are:

- **Text-based.** A text-based language offers a richer environment for people comfortable with using a keyboard. It is more representative of real-world programming than building software using icons and a mouse.
- **Forward-looking.** Skills developed while learning Snek should be transferable to other development environments.
- **Small.** This is not just to fit in smaller devices: the Snek language should be small enough to teach in a few hours to people with limited exposure to software.

Snek is Python-inspired, but it is not Python. It is possible to write Snek programs that run under a full Python system, but most Python programs will not run under Snek.

Chapter 2. A Gentle Snek Tutorial

Before we get into the details of the language, let's pause and just explore the language a bit to get a flavor of how it works. We won't be covering anything in detail, nor will all the subtleties be explored. The hope is to provide a framework for those details.

This tutorial shows what appears on the screen — both what the user types and what Snek displays. User input is shown **`in bold face, like this`** on the lines which start with > or +. Snek output is shown `in a lighter face, like this` on other lines.

2.1. Hello World

A traditional exercise in any new language is to get it to print the words `hello, world` to the console. Because Snek offers an interactive command line, we can actually do this in several ways.

The first way is to use Snek to echo back what you type at it. Start up Snek on your computer (perhaps by finding Snek in your system menu or by typing `snek` at the usual command prompt). When it first starts, Snek will introduce itself and then wait for you to type something.

```
Welcome to Snek version v1.3
>
```

At this `> ` prompt, Snek will print anything you type to it:

```
> 'hello, world'
'hello, world'
```

Here we see that Snek strings can be enclosed in single quotes. Strings can also be enclosed in double quotes, which can be useful if you want to include single quote marks in them. Snek always prints strings using single quotes, so the output here is the same as before.

```
> "hello, world"
'hello, world'
```

Snek is actually doing something a bit more complicated than echoing what you type.

What you are typing is called an “expression”, and Snek takes the expression, computes the value that it represents and prints that out. In this case, the value of either **`'hello, world'`** or **`"hello, world"`** is `'hello, world'`.

Stepping up a notch, instead of inputting **`'hello, world'`** directly, we can write a more complicated expression which computes it:

```
> 'hello,' + ' world'
'hello, world'
```

At this point, we’re using the feature of the interactive environment which prints out the value of expressions entered. Let’s try using the print function instead:

```
> print('hello, world')
hello, world
```

This time, Snek printed the string without quote marks. That’s because the print function displays exactly what it was given without quote marks while the command processor prints values in the same format as they would appear in a program (where you’d need the quote marks).

You might wonder where the value from evaluating the expression **`print('hello, world')`** is printed. After all, Snek printed the value of other expressions. The answer is that the `print` function evaluates to “no value”, and when Snek sees “no value”, it doesn’t print anything. We’ll see this happen several more times during the tutorial.

2.2. Variables

Variables are Snek’s way of remembering things. Each variable has a name, like `moss` or `tree`, and each variable can hold one. You set (or “assign”) the value of a variable using the = operator, and you get the value by using the name elsewhere:

```
> moss = 'hello, world'
> moss
'hello, world'
```

Snek creates a variable whenever you assign a value to it for the first time.

2.3. Functions

Let's define a function which uses `print` to print `hello world` and call it. To define a new function in Snek, we use the `def` keyword like this:

```
> def hello():
+     print('hello, world')
+
> hello()
hello, world
```

There's lots of stuff going on here. First, we see how to declare a function by using the `def` keyword, followed by the name of the function, followed by the “arguments” in parentheses. We'll talk about arguments in the next section, Simple Arithmetic. For now just type `()`. After the arguments there's a colon.

Colons appear in several places in Snek and (outside of dictionaries) are used in the same way. After a colon, Snek expects to see a list of statements. The usual way of including a list of statements is to type them, one per line, indented from the line containing the colon by a few spaces. The number of spaces doesn't matter, but each line has to use the same indentation. When you're done with the list of statements, you enter a line with the old indentation level.

While entering a list of statements, the command processor will prompt with + instead of > to let you know that it's still waiting for more input before it does anything. A blank line ends the list of statements for the `hello` function and gets you back to the regular command prompt.

Finally, we call the new `hello` function and see the results.

Snek normally ends each print operation by moving to the next line. That's because the print function has a named parameter called end which is set to a newline (`'\n'`) by default. You can change it to whatever you like, as in:

```
> def hello():
+     print('hello', end=',')
+     print(' world', end='\n')
+
> hello()
hello, world
```

The first call appends a `,` to the output, while the second call explicitly appends a newline character, causing the output to move to the next line. There are a few characters that use this backslash notation; those are described in the section on String Constants.

2.4. Simple Arithmetic

Let's write a function to convert from Fahrenheit temperatures to Celsius. If you recall, that's:

> °C = (5/9)(°F - 32)

Snek can't use the ° sign in variable names, so we'll just use C and F:

```
> # Convert from Fahrenheit to Celsius
> def f_to_c(F):
+     return (5/9) * (F - 32)
+
> f_to_c(38)
3.333333
```

The # character introduces a comment, which extends to the end of the line. Anything within a comment is ignored by Snek.

The `f_to_c` function takes one "argument" called F. Inside the function, F is a variable which is set to the value you place inside the parentheses when you call `f_to_c`. In this example, we're calling `f_to_c` with the value 38. Snek gets the value 38 from F whenever Snek finds it in the function:

```
+     return (5/9) * (F - 32)
⇒
      return (5/9) * (38 - 32)
⇒
      return 3.333333
```

Snek requires an explicit multiplication operator, *, as it doesn't understand the mathematical convention that adjacent values should be multiplied. The return statement is how we tell Snek that this function computes a value that should be given back to the caller.

Numbers in Snek may be written using _ as a separator, which is especially useful when writing large numbers.

```
> # you can write
> c = 299_792_458
> # and Snek will interpret as
> c = 299792458
```

2.5. Loops, Ranges and Printing Two Values

Now that we have a function to do this conversion, we can print a handy reference table for offline use:

```
> # Print a handy conversion table
> def f_to_c_table():
+   for F in range(0, 100, 10):
+     C = f_to_c(F)
+     print('%f F = %f C' % (F, C))
+
> f_to_c_table()
0.000000 F = -17.777779 C
10.000000 F = -12.222223 C
20.000000 F = -6.666667 C
30.000000 F = -1.111111 C
40.000000 F = 4.444445 C
50.000000 F = 10.000000 C
60.000000 F = 15.555556 C
70.000000 F = 21.111113 C
80.000000 F = 26.666668 C
90.000000 F = 32.222225 C
```

We see a new statement here: the `for` statement. This walks over a range of values, assigning the control variable (F, in this case) to each of the values in the range and then evaluating the list of statements within it. The `range` function creates the list of values for F by starting at the first value and stepping to just before the second value. If you give `range` only two arguments, Snek will step by 1. If you give `range` only one argument, Snek will use 0 as the starting point.

We need to insert the numeric values into the string shown by print. Many languages use a special formatted-printing function to accomplish this. In Snek, there's a more general-purpose mechanism called “string interpolation”. String interpolation uses the

% operator. Snek walks over the string on the left and inserts values from the list of values enclosed in parenthesis on the right wherever there is a % followed by a character. The result of string interpolation is another string which is then passed to print, which displays it.

How the values are formatted depends on the character following the % mark; that's discussed in the String Interpolation section. How to make that set of values on the right is discussed in the next section, Lists and Tuples

2.6. Lists and Tuples

Lists and Tuples in Snek are closely related data types. Both represent an ordered set of objects. The only difference is that Lists can be modified after creation while Tuples cannot. We call Lists "mutable" and Tuples "immutable". Lists are input as objects separated by commas and enclosed in square brackets, Tuples are input as objects separated by commas and enclosed in parentheses:

```
> [ 'hello,', ' world' ]
['hello,', ' world']
> ( 'hello,', ' world' )
('hello,', ' world')
```

Let's assign these to variables so we can explore them in more detail:

```
> l = [ 'hello,', ' world' ]
> t = ( 'hello,', ' world' )
```

As mentioned earlier, Lists and Tuples are ordered. That means that each element in a List or Tuple can be referenced by number. This number is called the index of the element, in Snek, indices start at 0:

```
> l[0]
'hello,'
> t[1]
' world'
```

Lists can be modified, Tuples cannot:

```
> l[0] = 'goodbye,'
> l
['goodbye,', ' world']
> t[0] = 'beautiful'
<stdin>:5 invalid type: ('hello,', ' world')
```

That last output is Snek telling us that the value ('hello', ' world') cannot be modified.

We can use another form of the `for` statement to iterate over the values in a List or Tuple:

```
> def print_list(list):
+     for e in list:
+         print(e)
+
> print_list(l)
goodbye,
 world
> print_list(t)
hello,
 world
```

Similar to the form described in the Loops, Ranges and Printing Two Values section, this `for` statement assigns the control variable (e in this case) to each of the elements of the list in turn and evaluates the statements within it.

Lists and Tuples can be concatenated (joined into a single thing) with the + operator:

```
> ['hello,'] + [' world']
['hello,', ' world']
```

Tuples of one element have a slightly odd syntax, to distinguish them from expressions enclosed in parentheses: the value within the Tuple is followed by a comma:

```
> ( 'hello' , ) + ( 'world' , )
('hello', 'world')
```

2.7. Dictionaries

Dictionaries are the fanciest data structure in Snek. Like Lists and Tuples, Dictionaries hold multiple values. Unlike Lists and Tuples, Dictionaries are not indexed by numbers. Instead, Dictionaries are indexed by another Snek value. The only requirement is that the index value be immutable, so that it can never change. Lists and Dictionaries are the only mutable data structures in Snek: anything else can be used as a Dictionary index.

The indexing value in a Dictionary is called the “key”, the indexed value is called the “value”. Dictionaries are input by enclosing key/value pairs, separated by commas, inside curly braces:

```
> { 1:2, 'hello,' : ' world' }
{ 'hello,':' world', 1:2 }
```

Note that Snek re-ordered our dictionary. That’s because Dictionaries are always stored in sorted order, and that sorting includes the type of the keys. Dictionaries can contain only one element with a given key: you’re free to specify dictionaries with duplicate keys, but only the last value will occur in the resulting Dictionary.

Let’s assign our Dictionary to a variable and play with it a bit:

```
> d = { 1:2, 'hello,' : ' world' }
> d[1]
2
> d['hello,']
' world'
> d[1] = 3
> d['goodnight'] = 'moon'
> d
{ 'goodnight':'moon', 'hello,':' world', 1:3 }
> d[56]
<stdin>:7 invalid value: 56
```

This example shows creating the Dictionary and assigning it to d, then fetching elements of the dictionary and assigning new values. You can add elements to a dictionary by using an index that is not already present. When you ask for an element which isn’t present, you get an error message.

You can also iterate over the keys in a Dictionary using the same `for` syntax used above. Let’s try our print_list function on d:

```
> print_list(d)
goodnight
hello,
1
```

You can test to see if an element is in a Dictionary using the `in` operator:

```
> if 1 in d:
+     print('yup')
+ else:
+     print('nope')
+
yup
> if 56 in d:
+     print('yup')
+ else:
+     print('nope')
+
nope
```

2.8. While

The `for` statement is useful when iterating over a range of values. Sometimes we want to use more general control flow. We can rewrite our temperature conversion chart program using a while loop as follows:

```
> def f_to_c_table():
+   F = 0
+   while F < 100:
+     C = f_to_c(F)
+     print('%f F = %f C' % (F, C))
+     F = F + 10
+
> f_to_c_table()
0.000000 F = -17.777779 C
10.000000 F = -12.222223 C
20.000000 F = -6.666667 C
30.000000 F = -1.111111 C
40.000000 F = 4.444445 C
50.000000 F = 10.000000 C
60.000000 F = 15.555556 C
70.000000 F = 21.111113 C
80.000000 F = 26.666668 C
90.000000 F = 32.222225 C
```

This does exactly what the for loop did in the Loops, Ranges and Printing Two Values section: it first assigns 0 to F, then iterates over the statements until F is no longer less than 100.

2.9. If

"If" statements provide a way of selecting one of many paths of execution. Each block of statements is preceded by an expression: if the expression evaluates to `True`, then the following statements are executed. Otherwise, the next test is tried until the end of the `if` is reached. Here's a function which measures how many upper case letters, lower case letters and digits are in a string:

```
> def count_chars(s):
+     d = 0
+     l = 0
+     u = 0
+     o = 0
+     for c in s:
+         if '0' <= c and c <= '9':
+             d += 1
+         elif 'a' <= c and c <= 'z':
+             l += 1
+         elif 'A' <= c and c <= 'Z':
+             u += 1
+         else:
+             o += 1
+     print('digits %d" % d)
+     print('lower %d" % l)
+     print('upper %d" % u)
+     print('other %d" % o)
+
> count_chars('4 Score and 7 Years Ago')
digits 2
lower 13
upper 3
other 5
```

The `elif` statements try other alternatives if previous `if` tests have not worked. The `else` statement is executed if all previous `if` and `elif` tests have not worked.

This example also introduces the less-than-or-equal comparison operator <= and demonstrates that `for v in a` also works on strings.

2.10. Controlling GPIOs

General-purpose IO pins, or “GPIOs”, are pins on an embedded processor which can be controlled by a program running on that processor.

When Snek runs on embedded devices like the Duemilanove or the Metro M0 Express, it provides functions to directly manipulate these GPIO pins. You can use either of these, or any other device which uses the standard Arduino pin numbers, for these examples.

2.10.1. Turning on the built-in LED

Let’s start by turning on the LED which is usually available on Digital pin 13:

```
> talkto(D13)
> on()
```

Let’s get a bit fancier and blink it:

```
> talkto(D13)
> while True:
+     onfor(.5)
+     time.sleep(.5)
```

2.10.2. Hooking up a digital input

Find a bit of wire to connect from Digital pin 1 to GND and let’s control the LED with this primitive switch:

```
> talkto(D13)
> while True:
+     if read(D1):
+         on()
+     else:
+         off()
```

When the wire is connected, the LED turns **off**, and when the wire is not, the LED turns **on**. That’s how simple switches work on Arduino.

Snek repeatedly reads the input and sets the LED as fast as it can. This happens thousands of times per second, giving the illusion that the LED changes the instant the switch changes.

2.10.3. Using an analog input

If you’ve got a light sensor or potentiometer, you can hook that up to Analog pin 0 and make the LED track the sensor:

```
> talkto(D13)
> while True:
+     onfor(1-read(A0))
+     time.sleep(1-read(A0))
```

2.10.4. Controlling motors

So far we’ve only talked about using one pin at a time. Arduino motor controllers take two pins: one for power and one for direction. Snek lets you tell it both pins at the same time and then provides separate functions to set the power and direction. If you have a motor controller hooked to your board with pin 3 as power and pin 2 as direction you can run the motor at half power and have it alternate directions with:

```
> talkto((3,2))
> setpower(0.5)
> on()
> while True:
+     setleft()
+     time.sleep(1)
+     setright()
+     time.sleep(1)
```

Snek Reference Manual

The remainder of this book is a reference manual for the Snek language, including built-in functions and the Snek development environment.

Chapter 3. Lexical Structure

Snek programs are broken into a sequence of tokens by a lexer. The sequence of tokens is recognized by a parser.

3.1. Numbers

Snek supports 32-bit floating point numbers and understands the usual floating point number format:

```
<integer><fraction><exponent>
123.456e+12
```

integer

A non-empty sequence of decimal digits

fraction

A decimal point (period) followed by a possibly empty sequence of decimal digits

exponent

The letter 'e' or 'E' followed by an optional sign and a non-empty sequence of digits indicating the exponent magnitude.

All parts are optional, although the number must include at least one digit in either the integer part or the fraction.

Floating point values (represented internally in IEEE 854 32-bit format) range from approximately `-1.70141e+38` to `1.70141e+38`. There is 1 sign bit, 8 bits of exponent and 23 stored/24 effective bits of significand (often referred to as the mantissa). There are two values of infinity (positive and negative) and a "Not a Number" (NaN) value indicating a failed computation. Computations using integer values will generate an error for values which cannot be represented as a 24-bit integer. That includes values that are too large and values with fractional components.

3.2. Names

Names in Snek are used to refer to variables, both global and local to a particular function. Names consist of an initial letter or underscore, followed by a sequence of letters, digits, underscore and period. Here are some valid names:

```
hello
_hello
_h4
math.sqrt
```

And here are some invalid names:

```
.hello
4square
```

3.3. Keywords

Keywords look like regular Snek names, but they are handled specially by the language and thus cannot be used as names. Here is the list of Snek keywords:

```
and       assert    break     continue
def       del       elif      else
for       global    if        import
in        is        not       or
pass      range     return    while
```

3.4. Punctuation

Snek uses many special characters to make programs more readable; separating out names and keywords from operators and other syntax.

```
:       ;       ,       (       )       [       ]       {
}       +       -       *       **      /       //      %
&       |       ~       !       ^       <<      >>      =
+=      -=      *=      **=     /=      //=     %=      &=
|=      ~=      ^=      <<=     >>=     >       !=      <
<=      ==      >=      >
```

3.5. White Space (Spaces and Newlines)

Snek uses indentation to identify program structure. Snek does not permit tabs to be used for indentation; tabs are invalid characters in Snek programs. Statements in the same block (list of statements) are indented the same amount; statements in deeper blocks are indented more, statements in shallower blocks less.

When typing Snek directly at the Snek prompt blank lines become significant, as Snek cannot know what you will type next. You can see this in the Tutorial, where Snek finishes an indented block at the blank line.

When loading Snek from a file, blank lines (and lines which contain only a comment) are entirely ignored; indentation of those lines doesn’t affect the block indentation level. Only lines with Snek tokens matter in this case.

Spaces in the middle of the line are only significant if they are necessary to separate tokens; you can insert as many or as few as you like in other places.

3.6. String Constants

String constants in Snek are enclosed in either single or double quotes. Use single quotes to easily include double quotes in the string, and vice-versa. Strings cannot span multiple lines, but you can input multiple strings adjacent to one another and they will be merged into a single string constant in the program.

\n

Newline. Advance to the first column of the next line.

\r

Carriage Return. Move to the first column on the current line.

\t

Tab. Advance to the next 'tab stop' in the output. This is usually the next multiple-of-8 column in the current line.

\xdd

Hex value. Use two hex digits to represent any character.

\

Backslash. Use two backslashes in the input to get one backslash in the string constant.

Anything else following the backslash is just that character. In particular:

\"

Literal double-quote. Useful inside double-quoted strings.

\'

Literal single-quote. Useful inside single-quoted strings.

3.7. List and Tuple Constants

List and Tuple constants in Snek are values separated by commas and enclosed in brackets: square brackets for Lists, parentheses for Tuples.

Here are some valid Lists:

```
[1, 2, 3]
['hello', 'world']
[12]
```

Here are some valid Tuples:

```
(1, 2, 3)
('hello', 'world')
(12,)
```

Note the last case — to distinguish between a value in parentheses and Tuple with one value, the Tuple needs to have a trailing comma. Only single-valued Tuples are represented with a trailing comma.

3.8. Dictionary Constants

Dictionaries in Snek are key/value pairs separated by commas and enclosed in curly braces. Keys are separated from values with a colon.

Here are some valid Dictionaries:

```
{ 1:2, 3:4 }
{ 'pi' : 3.14, 'e' : 2.72 }
{ 1: 'one' }
```

You can include entries with duplicate keys: the resulting Dictionary will contain only the last entry. The order of the entries does not matter otherwise: the resulting dictionary will always be the same:

```
> { 1:2, 3:4 } == { 3:4, 1:2 }
1
```

When Snek prints dictionaries, they are always printed in the same order, so two equal dictionaries will have the same string representation.

Chapter 4. Data Types

Like Python, Snek does not have type declarations. Instead, each value has an intrinsic representation — any variable may hold a value with any representation. To keep things reasonably simple, Snek has only a handful of representation types:

Numbers

Instead of having integers and floating point values, Snek represents numbers in floating point as described earlier. Integer values of less than 24 bits can be represented exactly in these floating point values: programs requiring precise integer behavior can still work as long as the values can be held in 24-bits.

Strings

Strings are just lists of bytes. Snek does not have any intrinsic support for encodings. Because they are just lists of bytes, you can store UTF-8 values in them comfortably. Just don’t expect indexing to return Unicode code points.

Lists

Lists are ordered collections of values. You can change the contents of a list by adding or removing elements. In other languages, these are often called arrays or vectors. Lists are “mutable” values.

Tuples

Tuples are immutable lists of values. You can’t change a tuple itself once it is created. If any *element* of the tuple *is* mutable, you can modify that element and see the changed results in the tuple.

Dictionaries

A dictionary is a mapping between **keys** and **values**. They work somewhat like Lists in that you can store and retrieve values in them. The index into a Dictionary may be any immutable value, which is any value other than a List or Dictionary or Tuple containing a List or Dictionary. Dictionaries are “mutable” values.

Functions

Functions are values in Snek. You can store them in variables or lists, and then fetch them later.

Boolean

Like Python, Snek doesn’t have an explicit Boolean type. Instead, a variety of values work in Boolean contexts as True or False values. All non-zero Numbers, non-empty Strings/Lists/Tuples/Dictionaries and all Functions are True. Zero, empty Strings/Lists/Tuples/Dictionaries are False. The name True is just another way of typing the number one. Similarly, the name False is just another way of typing the number zero.

4.1. Lists and Tuples

The += operator works a bit different on Lists than any other type — it appends to the existing list rather than creating a new list. This can be seen in the following example:

```
> a = [1,2]
> b = a
> a += [3]
> b
[1, 2, 3]
```

Compare this with Tuples, which (as they are immutable) cannot be appended to. In this example, b retains the original Tuple value. a gets a new Tuple consisting of `(3,)` appended to the original value.

```
> a = (1,2)
> b = a
> a += (3,)
> b
(1, 2)
> a
(1, 2, 3)
```

Chapter 5. Operators

Operators are things like + or –. They are part of the grammar of the language and serve to make programs more readable than they would be if everything was a function call. Like Python, the behavior of Snek operators often depends on the values they are operating on. Snek includes most of the Python operators. Some numeric operations work on floating point values, others work on integer values. Operators which work only on integer values convert floating point values to integers, and then take the integer result and convert back to a floating point value.

value* + *value

The Plus operator performs addition on numbers or concatenation on strings, lists and tuples.

value* – *value

The Minus operator performs subtraction on numbers.

value* * *value

The Multiplication operator performs multiplication on numbers. If you multiply a string, 's', by a number, 'n', you get 'n' copies of 's' concatenated together.

value* / *value

The Divide operator performs division on numbers.

value* // *value

The Div operator performs "integer division" on numbers, producing a result such that `x // y == floor(x / y)` for all numbers `x` and `y`.

value* % *value

The Modulus operator gives the "remainder after division" of its arguments, such that `x == y * (x // y) + x % y` for all numbers `x` and `y`. If the left operand is a string, it performs “interpolation” with either a single value or a list/tuple of values and is used to generate formatted output. See the String Interpolation section for details.

value* ** *value

The Power operator performs exponentiation on numbers.

value* & *value

The Logical And operator performs bit-wise AND on integers.

value* | *value

The Logical Or operator performs bit-wise OR on integers.

value* ^ *value

The Logical Xor operator performs bit-wise XOR on integers.

value* << *value

The Left Shift operator does bit-wise left shift on integers.

value* >> *value

The Right Shift operator does bit-wise left shift on integers.

! ***value***

The Not operator performs a Boolean negation operation on its right operand. That is, if the operand is one of the True values, then Not returns `False` (which is 0), and if the operand is a `False` value, then Not returns `True` (which is 1).

~ ***value***

The Logical Not operator performs a bit-wise NOT operation on its integer operand.

– ***value***

When used as a unary prefix operator, the Unary Minus operator performs negation on numbers.

+ ***value***

When used as a unary prefix operator, the Unary Plus operator does nothing at all to a number.

***value* [*index*]**

The Index operator selects the *index*-th member of strings, lists, tuples and dictionaries.

[*value [* , *value …]*]

The List operator creates a new List with the provided members. Note that a List of one value does not have any comma after the value and is distinguished from the Index operator solely by how it appears in the input.

(*value*)

Parenthesis serve to control the evaluation order within expressions. Values inside the parenthesis are computed before they are used as values for other operators.

(*value* ,) or (*value [, value …]*)

The Tuple operator creates a new Tuple with the provided members. A Tuple of one value needs a trailing comma so that it can be distinguished from an expression inside of parenthesis.

{ *key* : *value [, key : value …]* }

The Dictionary operator creates a new Dictionary with the provided key/value pairs. All of the *keys* must be immutable.

5.1. Slices

The Slice operator, *value* `[` *base* `:` *bound* `:` *stride* `]`, extracts a sequence of values from Strings, Lists and Tuples. It creates a new object with the specified subset of values from the original. The new object matches the type of the original.

base

The first element of *value* selected for the slice. If *base* is negative, then it counts from the end of *value* instead the beginning.

bound

The first element of *value* beyond the range selected for the slice.

stride

The spacing between selected elements. *Stride* may be negative, in which case elements are selected in reverse order, starting towards the end of *value* and working towards the beginning. It is an error for *stride* to be zero.

All three values are optional. The default value for *stride* is one. If *stride* is positive, the default value for *base* is 0 and the default for *bound* is the length of the array. If *stride* is negative, the default value for *base* is the index of the last element in *value* (which is `len(`*value*`) - 1`) and the default value for *bound* is −1. A slice with a single colon is taken as indicating *base* and *bound*. Here are some examples:

```
> # initialize a to a
> # Tuple of characters
> a = ('a', 'b', 'c', 'd', 'e', 'f')
> # With all default values, a[:] looks
> # the same as a
> a[:]
('a', 'b', 'c', 'd', 'e', 'f')
> # Reverse the Tuple
> a[::-1]
('f', 'e', 'd', 'c', 'b', 'a')
> # Select the end of the Tuple starting
> # at index 3
> a[3:]
('d', 'e', 'f')
> # Select the beginning of the Tuple,
> # ending before index 3
> a[:3]
('a', 'b', 'c')
```

5.2. String Interpolation

String interpolation in Snek can be confused with formatted printing in other languages. In Snek, the `print` function prints any arguments as they are given, separating them with spaces on the line. String interpolation produces a new String from a format specification String and a List or Tuple of parameters: this new String can be used for printing or for anything else one might want a String for.

If only a single value is needed, it need not be enclosed in a List or Tuple. Beware that if this single value is itself a Tuple or List, then String interpolation will get the wrong answer.

Within the format specification String are conversion specifiers which indicate where to insert values from the parameters. These are indicated with a `%` sign followed by a single character: this character is the format indicator and specifies how to format the value. The first conversion specifier uses the first element from the parameters, etc. The format indicator characters are:

`%d`
`%i`
`%o`
`%x`
`%X`

Format a number as a whole number, discarding any fractional part and without any exponent. `%d` and `%i` present the value in base 10. `%o` uses base 8 (octal) and `%x` and `%X` use base 16 (hexadecimal), with `%x` using lower case letters (a-f) and `%X` using upper case letters (A-F).

`%e`
`%E`
`%f`
`%F`
`%g`
`%G`

Format a number as floating point. The upper case variants use `E` for the exponent separator, lower case uses `e` and are otherwise identical. `%e` always uses exponent notation, `%f` never uses exponent notation. `%g` uses whichever notation makes the output smaller.

`%c`

Output a single character. If the parameter value is a number, it is converted to the character. If the parameter is a string, the first character from the string is used.

`%s`

Output a string. This does not insert quote marks or backslashes.

`%r`

Generate a printable representation of any value, similar to how the value would be represented in a Snek program.

If the parameter value doesn't match the format indicator requirements, or if any other character is used as a format indicator, then `%r` will be used instead.

Here are some examples of String interpolation:

```
> print('hello %s' % 'world')
hello world
> print('hello %r' % 'world')
hello 'world'
> print('pi = %d' % 3.1415)
pi = 3
> print('pi = %f' % 3.1415)
pi = 3.141500
> print('pi = %e' % 3.1415)
pi = 3.141500e+00
> print('pi = %g' % 3.1415)
pi = 3.1415
> print('star is %c' % 42)
star is *
> print('%d %d %d' % (1, 2, 3))
1 2 3
```

And here are a couple of examples showing why a single value may need to be enclosed in a Tuple:

```
> a = (1,2,3)
> print('a is %r' % a)
a is 1
> print('a is %r' % (a,))
a is (1, 2, 3)
```

In the first case, String interpolation is using the first element of a as the value instead of using all of a.

Chapter 6. Expression and Assignment Statements

value

An Expression statement simply evaluates *value*. This can be useful if *value* has a side-effect, like a function call that sets some global state. At the top-level, *value* is printed, otherwise it is discarded.

location* = *value

The Assignment statement takes the value on the right operand and stores it in the location indicated by the left operand. The left operand may be a variable, a list location or a dictionary location.

location* +=, −=, `=, /=, //=, %=,` *=, &=, |=, ^=, <<=, >>= *value

The Operation Assignment statements take the value of the left operand and the value of the right operand and performs the operation indicated by the operator. Then it stores the result back in the location indicated by the left operand. There are some subtleties about this which are discussed in the Lists and Tuples section of the Data Types chapter.

Chapter 7. Control Flow

Snek has a subset of the Python control flow operations, including trailing `else:` blocks for loops.

7.1. `if`

> `if` *value* `:` block *[* `elif` *value* `:` … *]* *[* `else:` block *]*

An If statement contains an initial `if` block, any number of `elif` blocks and then (optionally) an `else` block in the following structure:

```
if if_value :
    if statements
elif elif_value :
    elif_statements
…
else:
    else_statements
```

If *if_value* is true, then *if_statements* are executed. Otherwise, if *elif_value* is true, then *elif_statements* are executed. If none of the if or elif values are true, then the *else_statements* are executed.

7.2. `while`

> `while` *value* `:` block *[* `else:` block *]*

A While statements consists of a `while` block followed by an optional `else` block:

```
while while_value :
    block
else:
    block
```

While_value is evaluated and if it evaluates as `True`, the while block is executed. Then the system evaluates *while_value* again, and if it evaluates as `True` again, the while block is again executed. This continues until the *while_value* evaluates as `False`.

When the *while_value* evaluates as `False`, the `else:` block is executed. If a `break` statement is executed as a part of the while statements, then the program immediately jumps past the else statements. If a `continue` statement is executed as a part of the `while` statements, execution jumps back to the evaluation of *while_value*. The `else:` portion (with else statements) is optional.

7.3. `for`

> `for` *name* `in` *value* `:` block *[* `else:` block *]*

For each value v in the list of *values*, the `for` statement assigns v to *name* and then executes a block of statements. *Value* can be specified in two different ways: as a List, Tuple, Dictionary or String values, or as a range expression involving numbers:

```
for name in value:
    for statements
else:
    else statements
```

In this case, the *value* must be a List, Tuple, Dictionary or String. For Lists and Tuples, the values are the elements of the object. For Strings, the values are strings made from each separate (ASCII) character in the string. For Dictionaries, the values are the keys in the dictionary.

```
for name in range ( [ start , ] stop [ , step ] ):
    for statements
else:
    else statements
```

In this form, the `for` statement assigns a range of numeric values to *name*. Starting with *start*, and going while not beyond *stop*, *name* gets *step* added at each iteration. *Start* is optional; if not present, 0 will be used. *Step* is also optional; if not present, 1 will be used.

```
> for x in (1,2,3):
+     print(x)
+
1
2
3
> for c in 'hi':
+     print(c)
+
h
i
> a = { 1:2, 3:4 }
> for k in a:
+     print('key is %r value is %r' % (k, a[k]))
+
key is 1 value is 2
key is 3 value is 4
> for i in range(3):
+     print(i)
+
0
1
2
> for i in range(2, 10, 2):
+     print(i)
+
2
4
6
8
```

If a `break` statement is executed as a part of the `for` statements, then the program immediately jumps past the else statements. If a `continue` statement is executed as a part of the `for` statements, execution jumps back to the assignment of the next value to *name*. In both forms, the `else:` portion (with else statements) is optional.

7.4. `return` *value*

The Return statement causes the currently executing function immediately evaluate to *value* in the enclosing context.

```
> def r():
+     return 1
+     print('hello')
+
> r()
1
```

In this case, the `print` statement did not execute because the `return` happened before it.

7.5. break

The Break statement causes the closest enclosing `while` or `for` statement to terminate. Any optional `else:` clause associated with the `while` or `for` statement is skipped when the `break` is executed.

```
> for x in (1,2):
+     if x == 2:
+         break
+     print(x)
+ else:
+     print('else')
+
1
```

```
> for x in (1,2):
+     if x == 3:
+         break
+     print(x)
+ else:
+     print('else')
+
1
2
else
```

In this case, the first example does not print `else` due to the `break` statement execution rules. The second example prints `else` because the `break` statement is never executed.

7.6. `continue`

The `continue` statement causes the closest enclosing `while` or `for` statement to jump back to the portion of the loop which evaluates the termination condition. In `while` statements, that is where the *while_value* is evaluated. In `for` statements, that is where the next value in the sequence is computed.

```
> vowels = 0
> other = 0
> for a in 'hello, world':
+     if a in 'aeiou':
+         vowels += 1
+         continue
+     other += 1
+
> vowels
3
> other
9
```

The `continue` statement skips the execution of `other += 1`, otherwise `other` would be 12.

7.7. `pass`

The `pass` statement is a place-holder that does nothing and can be used any place a statement is needed when no execution is desired.

```
> if 1 != 2:
+     pass
+ else:
+     print('equal')
+
```

This example ends up doing nothing as the condition directs execution through the `pass` statement.

Chapter 8. Other Statements

8.1. `import` *name*

The Import statement is ignored and is part of Snek so that Snek programs can be run using Python.

```
> import curses
```

8.2. `global` *name [, name …]*

The Global statement marks the names as non-local; assignment to them will not cause a new variable to be created.

```
> g = 0
> def set_local(v):
+     g = v
+
> def set_global(v):
+     global g
+     g = v
+
>  set_local(12)
> g
0
> set_global(12)
> g
12
>
```

Because `set_local` does not include `global g`, the assignment to g creates a new local variable, which is then discarded when the function returns. `set_global` does include the `global g` statement, so the assignment to g references the global variable and the change is visible after that function finishes.

8.3. `del` *location*

The Del statement deletes either variables or elements within a List or Dictionary.

8.4. `assert` *value*

If *value* is `False`, the program will print `AssertionError` and then stop. Otherwise, the program will continue executing. This is useful to add checks inside your program to help catch problems earlier.

Chapter 9. Functions

Functions in Snek (as in any language) provide a way to encapsulate a sequence of operations. They can be used to help document what a program does, to shorten the overall length of a program or to hide the details of an operation from other parts of the program.

Functions take a list of “positional” parameters, then a list of “named” parameters. Positional parameters are all required, and are passed in the caller in the same order they appear in the declaration. Named parameters are optional; they will be set to the provided default value if not passed by the caller. They can appear in any order in the call. Each of these parameters is assigned to a variable in a new scope; variables in this new scope will hide global variables and variables from other functions with the same name. When the function returns, all variables in this new scope are discarded.

Additional variables in this new scope are created when they are assigned to, unless they are included in a `global` statement.

9.1. `def`

`def` *fname* `(` *pos1 [* `,` *posn …] [* `,` *namen* = *defaultn …]* `)` `:` block

A `def` statement declares (or re-declares) a function. The positional and named parameters are all visible as local variables while the function is executing.

Here’s an example of a function with two parameters:

```
> def subtract(a,b):
+     return a - b
+
> subtract(3,2)
1
```

And here’s a function with one positional parameter and two named parameters:

```
> def step(value, times=1, plus=0):
+     return value * times + plus
+
> step(12)
12
> step(12, times=2)
24
> step(12, plus=1)
13
> step(12, times=2, plus=1)
25
```

Chapter 10. Standard Built-in Functions

Snek includes a small set of standard built-in functions, but it may be extended with a number of system-dependent functions as well. This chapter describes the set of builtin functions which are considered a “standard” part of the Snek language and are provided in all Snek implementations.

10.1. `len(value)`

Len returns the number of characters for a String or the number of elements in a Tuple, List or Dictionary

```
> len('hello, world')
12
> len((1,2,3))
3
> len([1,2,3])
3
> len({ 1:2, 3:4, 5:6, 7:8 })
4
```

10.2. `print(` *`value1`* `, `` ` `` *`value2`*, …, `end='\n')`

Print writes all of its positional parameters to the console separated by spaces (`' '`) followed by the end named parameter (default: `'\n'`).

```
> print('hello world', end='.')
hello world.>
> print('hello', 'world')
hello world
>
```

10.3. `sys.stdout.flush()`

Flush output to the console, in case there is buffering somewhere.

10.4. `ord(` *string* `)`

Converts the first character in a string to its ASCII value.

```
>ord('A')
65
```

10.5. `chr(` *number* `)`

Converts an ASCII value to a one character string.

```
> chr(65)
'A'
```

10.6. `math.sqrt(` *number* `)`

Compute the square root of its numeric argument.

```
> math.sqrt(2)
1.414214
```

Chapter 11. Common System Functions

These functions are system-dependent, but are generally available. If they are available, they will work as described here.

11.1. `exit(` *value* `)`

Terminate Snek and return *value* to the operating system. How that value is interpreted depends on the operating system. On Posix-compatible systems, *value* should be a number which forms the exit code for the Snek process with zero indicating success and non-zero indicating failure.

11.2. `time.sleep(` *seconds* `)`

Pause for the specified amount of time (which can include a fractional part).

```
> time.sleep(1)
>
```

11.3. `time.monotonic()`

Return the time (in seconds) since some unspecified reference point in the system history. This time always increases, even if the system clock is adjusted (hence the name). Because Snek uses single-precision floating point values for all numbers, the reference point will be close to the starting time of the Snek system, so values may be quite small.

```
> time.monotonic()
6.859814
```

11.4. `random.seed(` *seed* `)`

Re-seeds the random number generator with `seed`. The random number generator will always generate the same sequence of numbers if started with the same `seed`.

```
> random.seed(time.monotonic())
>
```

11.5. `random.randrange(` *max* `)`

Generates a random integer between 0 and max-1 inclusive.

```
> random.randrange(10)
3
```

Chapter 12. Math Functions

The Snek math functions offer the same functions as the Python math package, although at single precision instead of double precision. These functions are optional, but if any are provided, all are provided and follow the definitions here.

12.1. Number-theoretic and representation functions

math.ceil(x)

Return the ceiling of x, the smallest integer greater than or equal to x.

math.copysign(x,y)

Return a number with the magnitude (absolute value) of x but the sign of y.

math.fabs(x)

Return the absolute value of x.

math.factorial(x)

Return the factorial of x.

math.floor(x)

Return the floor of x, the largest integer less than or equal to x.

math.fmod(x,y)

Return the modulus of x and y: x - trunc(x/y) * y.

math.frexp(x)

Returns the normalized fraction and exponent in a tuple (frac, exp). 0.5 ≤ abs(frac) < 1, and x = frac * pow(2,exp).

math.fsum(l)

Returns the sum of the numbers in l, which must be a list or tuple.

math.gcd(x,y)

Return the greatest common divisor of x and y.

math.isclose(x,y,rel_val=1e-6,abs_val=0.0)

Returns a boolean indicating whether x and y are 'close' together. This is defined as abs(x-y) ≤ max(rel_tol * max(abs(a), abs(b)), abs_tol).

math.isfinite(x)

Returns True if x is finite else False.

math.isinf

Returns True if x is infinite else False.

math.isnan

Returns True if x is not a number else False.

math.ldexp(x,y)

Returns x * pow(2,y).

math.modf(x)

Returns (x - trunc(x), trunc(x)).

math.remainder(x,y)

Returns the remainder of x and y: x - round(x/y) * y.

math.trunc

Returns the truncation of x, the integer closest to x which is no further from zero than x.

round(x)

Returns the integer nearest x, with values midway between two integers rounding away from zero.

12.2. Power and logarithmic functions

math.exp(x)

Returns pow(e,x).

math.expm1(x)

Returns exp(x)-1.

math.exp2(x)

Returns pow(2,x).

math.log(x)

Returns the natural logarithm of x.

math.log1p(x)

Returns log(x+1).

math.log2(x)

Returns the log base 2 of x.

math.log10(x)

Returns the log base 10 of x.

math.pow(x,y)

Returns x raised to the yth power.

12.3. Trigonometric functions

math.acos(x)

Returns the arc cosine of x in the range of $0 \le acos(x) \le \pi$.

math.asin(x)

Returns the arc sine of x in the range of $-\pi/2 \le asin(x) \le \pi/2$.

math.atan(x)

Returns the arc tangent of x in the range of $-\pi/2 \le atan(x) \le \pi/2$.

math.atan2(y,x)

Returns the arc tangent of y/x in the range of $-\pi \le atan2(y,x) \le \pi$.

math.cos(x)

Returns the cosine of x.

math.hypot(x,y)

Returns sqrt(x*x + y*y).

math.sin(x)

Returns the sine of x.

math.tan(x)

Returns the tangent of x.

12.4. Angular conversion

math.degrees(x)

Returns x * 180/π.

math.radians(x)

Returns x * π/180.

12.5. Hyperbolic functions

math.acosh(x)

Returns the inverse hyperbolic cosine of x.

math.asinh(x)

Returns the inverse hyperbolic sine of x.

math.atanh(x)

Returns the inverse hyperbolic tangent of x.

math.cosh(x)

Returns the hyperbolic cosine of x: (exp(x) + exp(-x)) / 2.

math.sinh(x)

Returns the hyperbolic sine of x: (exp(x) - exp(-x)) / 2.

math.tanh(x)

Returns the hyperbolic tangent of x: sinh(x) / cosh(x).

12.6. Special functions

math.erf(x)

Returns the error function at x.

math.erfc(x)

Returns the complement of the error function at x. This is 1 - erf(x).

math.gamma(x)

Returns the gamma function at x.

math.lgamma(x)

Returns log(gamma(x)).

12.7. Mathematical constants

math.pi

The mathematical constant π, to available precision.

math.e

The mathematical constant e, to available precision.

math.tau

The mathematical constant τ, which is 2π, to available precision.

math.inf

The floating point value which represents ∞.

math.nan

The floating point value which represents Not a Number.

Chapter 13. GPIO Functions

On embedded devices, Snek has a range of functions designed to make manipulating the GPIO pins convenient. Snek keeps track of two pins for output and one pin for input. The two output pins are called Power and Direction. Each output function specifies which pins it operates on. All input and output values range between 0 and 1. Digital pins use only 0 or 1, analog pins support the full range of values from 0 to 1.

Input pins can be set so that they read as 0 or 1 when nothing is connected by using `pulldown` or `pullup`. Using `pullnone` makes the pin “float” to provide accurate analog readings. Digital pins are to `pullup` by default, Analog pins are set to `pullnone`.

Output pins are either **on** or **off**. A pin which is **on** has its value set to the current power for that pin; changes to the current power for the pin are effective immediately. A pin which is **off** has its output set to zero, but Snek remembers the `setpower` level and will restore the pin to that level when it is turned **on**.

13.1. `talkto(` *pin* `)`

Set the current output pins. If *pin* is a number, this sets both the Power and Direction pins. If *pin* is a List or Tuple, then the first element sets the Power pin and the second sets the Direction pin.

13.2. `setpower(` *power* `)`

Sets the power level on the current Power pin to *power*. If the Power pin is currently **on**, then this is effective immediately. Otherwise, Snek remembers the desired power level and will use it when the pin is turned **on**. Values less than zero set the power to zero, values greater than one set the power to one.

13.3. `setleft()`

Turns the current Direction pin **on**.

13.4. `setright()`

Turns the current Direction pin **off**.

13.5. `on()`

Turns the current Power pin **on**.

13.6. `off()`

Turns the current Power pin **off**.

13.7. `onfor(` *seconds* `)`

Turns the current Power pin **on**, delays for *seconds* and then turns the current Power pin **off**.

13.8. `read(` *pin* `)`

Returns the value of *pin*. If this is an analog pin, then `read` returns a value from `0 to 1` (inclusive). If this a digital pin, then `read` returns either `0` or `1`.

13.9. `pullnone(` *pin* `)`

Removes any `pullup` or `pulldown` settings for *pin*, leaving the value floating when nothing is connected. Use this setting on analog pins to get continuous values rather than just 0 or 1. This is the default setting for Analog pins.

13.10. `pullup(` *pin* `)`

Assigns a `pullup` setting for *pin*, so that the `read` will return 1 when nothing is connected. When in this mode, analog pins will return only 0 or 1. This is the default setting for Digital pins.

13.11. `pulldown(` *pin* `)`

Assigns a `pullup` setting for *pin*, so that the `read` will return 0 when nothing is connected. When in this mode, analog pins will return only 0 or 1. Note that some boards do not support this mode, in which case this function will not be available.

13.12. `stopall()`

Turns all pins off.

13.13. `neopixel(` *pixels* `)`

Programs either a set of neopixel devices connected to the current Power pin (when Power and Direction are the same) or a set of APA102 devices connected to the current Power (used for APA102 Data) and Direction (used for APA102 Clock) pins (when Power and Direction are different). *pixels* is a list or tuple, each element of which is a list or tuple of three numbers ranging from 0 to 1 for the desired red, green and blue intensity of the target neopixel.

```
> talkto(NEOPIXEL)
> pixels = [(0.33, 0, 0), (0, 0.66, 0), (0, 0, 1)]
> neopixel(pixels)
```

This example programs three NeoPixel devices, the first one is set to one third intensity red, the second to two thirds intensity green and the last to full intensity blue. If there are additional neopixel devices connected, they will not be modified. If there are fewer devices connected than the data provided, the extra values will be ignored.

Chapter 14. EEPROM built-in functions

Snek on embedded devices may include persistent storage for source code. This code is read at boot time, allowing boards with Snek loaded to run stand-alone. These functions are used by Snekde to get and put programs to the device.

14.1. `eeprom.write()`

Reads characters from the console and writes them to eeprom until a `^D` character is read.

14.2. `eeprom.show()`

Dumps the current contents of eeprom out to the console. If a parameter is passed to this function then a `^B` character is sent before the text, and a `^C` is sent afterwards. Snekde uses this feature to accurately capture the program text when the Get command is invoked.

14.3. `eeprom.load()`

Re-parses the current eeprom contents, just as Snek does at boot time.

14.4. `eeprom.erase()`

Erase the eeprom.

14.5. `reset()`

Restart the Snek system, erasing all RAM contents. As part of the restart process, Snek will re-read any source code stored in eeprom.

Chapter 15. Temperature Conversion Function

This function is included in devices that have a built-in temperature sensor.

15.1. `temperature(` *sensorvalue* `)`

The conversion function is pre-set with the parameters needed to convert from the temperature sensor value to degrees Celsius.

Chapter 16. Curses built-in functions

Curses provides a simple mechanism for displaying text on the console. The API is designed to be reasonably compatible with the Python curses module, although it is much less flexible. Snek only supports ANSI terminals, and doesn't have any idea what the dimensions of the console are. Not all Snek implementations provide the curses functions.

16.1. `curses.initscr()`

Puts the console into "visual" mode. Disables echo. Makes `stdscr.getch()` stop waiting for newline.

16.2. `curses.endwin()`

Resets the console back to "normal" mode. Enables echo. Makes `stdscr.getch()` wait for newlines.

16.3. `curses.noecho()`, `curses.echo()`, `curses.cbreak()`, `curses.nocbreak()`

All four of these functions are no-ops and are part of the API solely to make it more compatible with Python curses.

16.4. `stdscr.nodelay(` *nodelay* `)`

If *nodelay* is True, then `stdscr.getch()` will return -1 if there is no character waiting. If *nodelay* is False, the `stdscr.getch()` will block waiting for a character to return.

16.5. `stdscr.erase()`

Erase the screen.

16.6. `stdscr.addstr(` *row* , *column* , *string* `)`

Displays *string* at *row*, *column*. *Row* `0` is the top row of the screen. *Column* `0` is the left column. The cursor is left at the end of the string.

16.7. `stdscr.move(` *row* `,` *column* `)`

Moves the cursor to *row*, *column* without displaying anything there.

16.8. `stdscr.refresh()`

Flushes any pending screen updates.

16.9. `stdscr.getch()`

Reads a character from the console input. Returns a number indicating the character read, which can be converted to a string using `chr(c)`. If `stdscr.nodelay(nodelay)` was most recently called with *nodelay* `= True`, then `stdscr.getch()` will immediately return -1 if no characters are pending.

Chapter 17. Snek Development Environment

The Snek Development Environment, Snekde, is a Python program which runs on Linux, Mac OS X and Windows to work with small devices running Snek, such as the Duemilanove and Metro M0 Express boards.

17.1. Starting Snekde

On Windows and Linux, launch `snekde` from your application menu. On Mac OS X, Snekde is installed along with the other Snek files in the Snek folder inside your personal Applications folder, which is inside your Home folder. Double click on the Snekde icon to launch.

Snekde runs inside a console or terminal window and doesn’t use the mouse at all, instead it is controlled entirely using keyboard commands.

Snekde splits the window into two panes. The upper pane is the ”editor pane” that holds your Snek program. The lower pane is the “console pane” and handles communications with the Snek device.

17.2. Basic Navigation

Across the top of the window you’ll see a list of commands which are bound to function keys. Those are there to remind you how to control Snekde.

If your function keys don’t work, you can use the Esc key along with a number key instead. Press and release the Esc key, then press and release a number key. For instance, to invoke the F1 command, press and release Esc, then press and release '1'.

Between the two panes is a separator line. At the end of that line is the name of the currently connected Snek device, such as `/dev/ttyUSB0` on Linux or `COM12` on Windows. If there isn’t a device connected, it will say "<no device>".

The cursor shows which pane you are currently working with. To switch between the editor and console panes, use the F7 key. If you don’t have one of these, or if it doesn’t work, you can also use Esc-7 or Ctrl-o (press and hold the Ctrl key, press the `o` key and then release both).

You can move around the current pane with the arrow, home, end and page-up/page-down keys. Cut/paste/copy use Ctrl-x, Ctrl-v and Ctrl-c or Esc-x, Esc-v and Esc-c respectively. To mark a section of text for a Cut or Paste command, press Esc-space or Ctrl-space then use regular movement commands. The selected region of text will be highlighted.

17.3. Connecting to a Device

To connect to a device running Snek, press the F1 key (usually right next to the ESC key on your keyboard). That will display a dialog box in the middle of the screen listing all of the devices which might be running Snek (if you've got a serial modem or other similar device, that will also be listed here). Select the target device and press the ENTER key.

Don't expect anything to happen in the lower pane just yet; you'll have to get the attention of the device first.

Switch to the Console pane (F7) and press Ctrl-c to interrupt any currently running Snek program. You should see the Snek prompt ("> ") appear in the pane.

17.4. Getting and Putting Programs to a Device

The Snek device holds one program in non-volatile memory. When it starts up, it will run that program automatically. This lets you set up the device so that it will perform some action when it is turned on without needing to communicate with it first.

The Get command fetches the current program from the connected device and puts it into the Editor pane. The Put command writes the Editor pane contents into non-volatile memory in the target device and then restarts the target device to have it reload the program. Both of these commands will interrupt any running Snek program before doing any work.

17.5. Loading and Saving Programs to the Host

You can also save and load programs to the host file system. Both of these commands prompt for a filename using a file dialog. At the top of the dialog is the filename to use. The rest of the dialog contains directories and files within the same directory as the filename. Directories are enclosed in `[ ]`.

Using the arrow keys replaces the filename with the highlighted name. You can also edit the filename using backspace and entering a new name.

Select a filename by pressing enter. If the name is a directory, then the contents of that directory will replace the list of directories and files in the dialog. If the name is a file, then that will be used for the load or save operation.

To quit from the dialog and skip loading or saving a file, press Escape.

Appendix A: Snek on snekboard

Snek for the snekboard includes the Common System, Math, GPIO (including the `neopixel` function) and EEPROM fountain's. Snek for the snekboard provides pre-defined variables for the eight analog I/O pins as well as the four 9V motor controllers:

A1-A8

Analog input and output pins. When used as output pins, you can use setpower to control the drive power. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 3.3V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

M1-M4

Bi-directional 9V DC motor control, 2.5A max current. These are tuples with two values each. `M1[0]`, `M2[0]`, `M3[0]` and `M4[0]` are the power pins. `M1[1]`, `M2[1]`, `M3[1]` and `M4[1]` are the direction pins. Note that there's a bit of firmware behind these pins that keeps the outputs from changing power too rapidly.

NEOPIXEL

The two APA102 devices on the board, which can be driven using the `neopixel` function.

Snekboard includes a boot loader which presents as a USB mass storage device with a FAT file system. You can get the board into this mode by connecting the board to your computer over USB and then pressing the blue reset button twice in quick succession.

Then, find the `snek-board-1.3.uf2` file included in the Snek package for your machine and copy it to the `CURRENT.UF2` file to the snekboard file system.

Appendix B: Snek on Arduino Duemilanove

Snek for the Duemilanove includes the Common System, EEPROM, and GPIO functions. It does not include the Math functions, nor the `pulldown` function. Snek for the Duemilanove provides pre-defined variables for all of the GPIO pins:

D0 - D13

Digital input and output pins. By default, when used as input pins, Snek applies a pull-up resistor to the pin so that a disconnected pin will read as 1. Change this using `pullnone` or `pullup` functions.

A0 - A5

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 5V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone` or `pullup` functions.

Snek fills the ATMega 328P flash completely leaving no space for the usual serial boot loader, so installing Snek requires a programming puck, such as the USBTiny device.

On Linux, the Snek installation includes a shell script, snek-duemilanove-install, to install the binary using 'avrdude'. Read the snek-duemilanove-install manual (also included in the installation) for more information.

On other hosts, you'll need to install 'avrdude'. Once you've done that, there are two steps to getting Snek installed on the device.

1. Set the 'fuses' on the target device. This sets the start address back to the beginning of memory instead of the boot loader, and then has the device leave the eeprom contents alone when re-flashing. That means you won't lose your Snek program when updating the firmware.

```
$ avrdude -F -V -c usbtiny -p ATMEGA328P -U lfuse:w:0xff:m -U
hfuse:w:0xd7:m -U efuse:w:0xfd:m
```

2. Install the Snek binary.

```
$ avrdude -F -V -c usbtiny -p ATMEGA328P -U flash:w:snek-
duemilanove-1.3.hex
```

Appendix C: Snek on Adafruit ItsyBitsy and the Crowd Supply µduino

Snek for the ItsyBitsy and µduino includes the Common System, GPIO (without the `neopixel` function), and EEPROM functions. Snek for the itsybitsy provides pre-defined variables for all of the the GPIO pins:

D0 - D13

Digital input and output pins. By default, when used as input pins, Snek applies a pull-up resistor to the pin so that a disconnected pin will read as 1. Change this using `pullnone`, `pullup` or `pulldown` functions.

A0 - A5

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to either 3.3V (on the 3v device) or 5V (on the 5V device). By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

MISO, MOSI, SCK

Additional digital input and output pins. These work just like D0-D13. These are not present on the µduino board.

Snek fills the ATMega 32u4 flash completely leaving no space for the usual USB boot loader, so installing Snek requires a programming puck, such as the USBTiny device.

On Linux, the Snek installation includes shell scripts, snek-itsybitsy-install and snek-uduino-install which install the binary using 'avrdude'. Read the snek-itsybitsy-install or snek-uduino-install manual (also included in the installation) for more information.

The µduino programming wires are only available while the device is still connected to the carrier board. Normally the µduino has been broken off of that during manufacturing.

On other hosts, you’ll need to install 'avrdude'. Once you’ve done that, there are two steps to getting Snek installed on the device.

1. Set the 'fuses' on the target device. This sets the start address back to the beginning of memory instead of the boot loader, and then has the device leave the eeprom contents alone when re-flashing. That means you won’t lose your Snek program when updating the firmware.

```
$ avrdude -F -V -c usbtiny -p m32u4 -U lfuse:w:0xff:m -U
hfuse:w:0x91:m -U efuse:w:0xfd:m
```

2. Install the Snek binary. Pick the version for your board as that also sets the right clock speed. For 5v boards, install the 5v binary:

   ```
   $ avrdude -F -V -c usbtiny -p m32u4 -U flash:w:snek-itsybitsy5v-
   1.3.hex
   ```

   for 3v boards, use the 3v binary.

   ```
   $ avrdude -F -V -c usbtiny -p m32u4 -U flash:w:snek-itsybitsy3v-
   1.3.hex
   ```

   for µduino boards, use the µduino binary.

   ```
   $ avrdude -F -V -c usbtiny -p m32u4 -U flash:w:snek-uduino-1.3.hex
   ```

Appendix D: Snek on Adafruit ItsyBitsy M0

Snek for the Adafruit ItsyBitsy includes the Common System, Math, GPIO (including the `neopixel` function), and EEPROM functions. Snek for the itsybitsy m0 provides pre-defined variables for all of the the GPIO pins:

D0 - D13

Digital input and output pins. By default, when used as input pins, Snek applies a pull-up resistor to the pin so that a disconnected pin will read as 1. Change this using `pullnone`, `pullup` or `pulldown` functions. D5 on the ItsyBitsy M0 is hooked to a 3.3V to 5V converter so that it can drive 5V devices. This means it cannot be used as an input pin.

A0 - A5

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 3.3V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

SDA, SCL, MISO, MOSI, SCK

Additional digital input and output pins. These work just like D0-D13.

NEOPIXEL

The APA102 device on the board, which can be driven using the `neopixel` function.

The Adafruit ItsyBitsy M0 board includes a boot loader which presents as a USB mass storage device with a FAT file system. You can get the board into this mode by connecting the board to your computer over USB and then pressing the reset button twice in succession. In boot loader mode, the red LED on D13 will pulse rapidly for a few seconds, then more slowly. At that point, the APA102 device will turn green.

Once the ItsyBitsy M0 is in boot loader mode and has been mounted, find the `snek-itsybitsym0-1.3.uf2` file included in the Snek package for your machine and copy it to the `CURRENT.UF2` file on the ItsyBitsy M0 file system.

Appendix E: Snek on Arduino Mega

Snek for the Mega includes the Common System, EEPROM, GPIO (not including the `pulldown` function) and math functions. Snek for the Mega provides pre-defined variables for all of the GPIO pins:

D0-D53

Digital input and output pins. By default, when used as input pins, Snek applies a pull-up resistor to the pin so that a disconnected pin will read as 1. Change this using `pullnone` or `pullup` functions.

A0-A15

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 5V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone` or `pullup` functions.

Snek fits comfortably in the ATmega2560 flash, leaving plenty of space for the serial boot loader, so re-installing Snek can be done over USB. However, the default firmware loaded on the ATMega16u2 that acts as USB to serial converter doesn’t do any XON/XOFF flow control and so that should be replaced before installing Snek as Snekde will not get or put source code successfully without it.

On Linux, the Snek installation includes a shell script, snek-mega-install, to install the binary using 'avrdude'. Read the snek-mega-install manual (also included in the installation) for more information.

On other hosts, you’ll need to install 'avrdude'. Once you’ve done that, you can use it to get Snek installed on the device. Because the EEPROM fuse bit can’t be set this way, when you do this any Snek program stored on the device will be erased. Find out what port the Mega is connected to, use that as the value for `<port>` and then run 'avrdude' as follows:

```
$ avrdude -patmega2560 -cwiring -P<port> -b115200 -D -U flash:w:snek-
mega-1.3.hex:i
```

Appendix F: Snek on Metro M0 Express

Snek for the Metro M0 Express includes the Common System, Math, GPIO (including the `neopixel` function), and EEPROM functions. Snek for the metro m0 provides pre-defined variables for all of the GPIO pins:

D0 - D13

Digital input and output pins. By default, when used as input pins, Snek applies a pull-up resistor to the pin so that a disconnected pin will read as 1. Change this using `pullnone`, `pullup` or `pulldown` functions.

A0 - A5

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 3.3V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

SDA, SCL

Additional Digital input and output pins. These work just like D0-D13.

NEOPIXEL

The NeoPixel device installed on the board.

The Adafruit Metro M0 Express board includes a boot loader which presents as a USB mass storage device with a FAT file system. You can get the board into this mode by connecting the board to your computer over USB and then pressing the reset button twice in quick succession.

Then, find the `snek-metrom0-1.3.uf2` file included in the Snek package for your machine and copy it to the `CURRENT.UF2` file on the Metro M0 file system.

Appendix G: Snek on Feather M0 Express

Snek for the Feather M0 Express includes the Common System, Math, GPIO (including the `neopixel` function), and EEPROM functions. Snek for the feather provides pre-defined variables for all of the GPIO pins:

D0 - D13

Digital input and output pins. By default, when used as input pins, Snek applies a pull-up resistor to the pin so that a disconnected pin will read as 1. Change this using `pullnone`, `pullup` or `pulldown` functions.

A0 - A5

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 3.3V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

SDA, SCL, SCK, MOSI, MISO

Additional Digital input and output pins. These work just like D0-D13.

NEOPIXEL

The NeoPixel device installed on the board, which is connected to D8.

RX, TX

RX is D0, TX is D1.

The Adafruit Feather M0 Express board includes a boot loader which presents as a USB mass storage device with a FAT file system. You can get the board into this mode by connecting the board to your computer over USB and then pressing the reset button twice in quick succession.

Then, find the `snek-feather-1.3.uf2` file included in the Snek package for your machine and copy it to the `CURRENT.UF2` file on the Feather M0 file system.

Appendix H: Snek on Adafruit Crickit

Snek for the Crickit includes the Common System, Math, GPIO (including the `neopixel` function), and EEPROM functions. Snek for the Crickit provides names for all of the GPIO pins:

DRIVE1 - DRIVE4

High current "Darlington" 500mA drive outputs.

MOTOR1, MOTOR2

Bi-directional DC motor control, 1A max current. These are tuples with two values each. `MOTOR1[0]` and `MOTOR2[0]` are the power pins. `MOTOR1[1]` and `MOTOR2[1]` are the direction pins. Note that there's a bit of firmware behind these pins as the TI DRV8833 chip has a slightly funky control mechanism.

SERVO1 - SERVO4

Digital pins with PWM output

CAP1 - CAP4

Digital pins labeled "Capacitive Touch" on the Crickit board.

SIGNAL1 - SIGNAL8

The Signal pins. These provide digital output and analog input. SIGNAL5 - SIGNAL8 also provide PWM output

NEOPIXEL

The single NeoPixel device installed on the board.

NEOPIXEL1

The external NeoPixel connector.

The Adafruit Crickit board includes a boot loader which presents as a USB mass storage device with a FAT file system. You can get the board into this mode by connecting the board to your computer over USB and then pressing the reset button twice in quick succession.

Then, find the `snek-crickit-1.3.uf2` file included in the Snek package for your machine and copy it to the `CURRENT.UF2` file on the Crickit file system.

Appendix I: Snek on Adafruit Circuit Playground Express

Snek for the Circuit Playground Express includes the Common System, Math, GPIO (including the `neopixel` function), Temperature and EEPROM functions. Snek for the Playground provides names for all of the external connections as well as the built-in devices:

A0 - A7
: External GPIO connections, labeled around the perimeter of the board.

A8 or LIGHT
: Internal ambient light sensor. Returns a value indicating how much light is shining on the sensor.

A9 or TEMP
: Internal temperature sensor. Use the builtin `temperature` function to convert values read from this pin to degrees Celsius.

D4 or BUTTONA
: Connected to the momentary button labeled 'A'. 0 if not pressed, 1 if pressed.

D5 or BUTTONB
: Connected to the momentary button labeled 'B'. 0 if not pressed, 1 if pressed.

D7 or SWITCH
: Connected to the slide switch. 0 if slid right (towards the microphone), 1 if slid left (towards the speaker).

D13 or LED
: The red LED near the USB connector.

D8 or NEOPIXEL
: The string of 10 NeoPixel devices on the board.

The Adafruit Circuit Playground Express board includes a boot loader which presents as a USB mass storage device with a FAT file system. You can get the board into this mode by connecting the board to your computer over USB, sliding the switch to the right (towards the microphone) and then pressing the reset button twice in quick succession.

Then, find the `snek-playground-1.3.uf2` file included in the Snek package for your machine and copy it to the `CURRENT.UF2` file on the Circuit Playground Express file system.

Appendix J: Snek on Arduino SA Nano 33 IoT

Snek for the Nano 33 IoT includes the Common System, Math, GPIO, and EEPROM functions. Snek for the Nano 33 IoT provides names for all of the GPIO pins:

D0 - D12

Digital outputs By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

D13 or LED

The yellow LED near the USB connector.

A0-A5

Analog input and Digital output pins. When used as input pins, Snek will return a value from 0-1 indicating the ratio of the pin voltage to 3.3V. By default, when used as input pins, Snek does not apply either a pull-up or pull-down resistor to the pin so that a disconnected pin will read an indeterminate value. Change this using `pullnone`, `pullup` or `pulldown` functions.

The Arduino SA Nano 33 IoT board includes a boot loader which works with the Arduino IDE. Snek includes a replacement boot loader which presents as a USB mass storage device with a FAT file system. To install this boot loader, start the Arduino IDE, find the update-bootloader-nano33iot.ino project included in thte Snek package for your machine and load it into the Arduino IDE. Then compile and download that to the Nano 33 IoT board. That will replace the boot loader and restart the board, at which point it should present a file system. For future updates, you can get the board back into this mode by connecting the board to your computer over USB and then pressing the reset button twice in quick succession.

Onec the board is showing a file system on your computer, find the `snek-nano33iot-1.3.uf2` file included in the Snek package for your machine and copy it to the file system.

Index