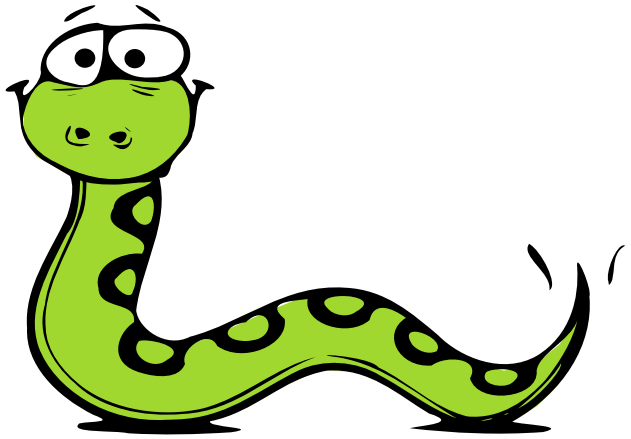

The Snek Programming Language

A Python-inspired Embedded Computing Language

Keith Packard <keithp@keithp.com

Version v0.8, 2019-03-04

Table of Contents

License

Acknowledgments

Thanks to Jane Kenny-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 teach, 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 both helped teach Arduino programming using Lego robots. Christopher has also been helping design and test Snek.

Keith Packard
keithp@keithp.com
http://keithp.com

Chapter 1. History and Motivations

Teaching computer programming to students in the 10-14 age range offers some interesting challenges. Introductory, "blocks", languages can become frustratingly slow to create code, and don't provide any concrete skills to bring to more advanced languages. Sophisticated languages like C, Java and even Python are so large as to overwhelm the student with rich semantics like objects and higher level programming constructs.

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

Because computing is so ubiquitous these days, introductory programming is taught today in a huge range of environments, from embedded systems through cloud-based systems. Many of these are technological dead-ends; closed systems that offer no way to even extract source code and re-use it in another environment.

Some systems, such as Raspberry PI and Arduino are open and allow developers to share code. However, while the smallest of these are similar in memory and CPU size to those early machines, they are programmed as embedded computers using a full C++ compiler running on a 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. However, students struggled with the complex syntax, especially typing the obscure punctuation marks and remembering to insert semicolons. The lack of any interactive mode made experimenting a bit slower than other systems.

After a couple of years, I built some custom Arduino hardware for our needs — I used screw terminals for all of the inputs and outputs, stuck a battery pack on the back and included four high-current H-bridge motor controllers to help animate the robots. They're still Arduinos though, there's an ATmega 328P processor and a FTDI USB to serial converter, so we were able to use the stock Arduino development tools.

There have been students who got past the obstacles and figured out how to use them:

- 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 height-measuring robot to automatically measure people's height. This used an optical sensor to monitor movement of a sensor and communication with an attached 7-segment display. We've shown this device at numerous local Lego shows.

However, other students dreaded having to use the Arduino systems with complaints about "too much typing" and "why is it so picky about semicolons".

The hardware was just what we wanted, but the software was not aimed at young students just starting to write code.

1.2. A New Language

Instead of throwing out our existing systems and starting over, I wondered if we couldn't keep using the same (hand-made) hardware and just change the programming environment.

So 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, but that is still a large language which takes a few hundred kB of ROM and a significant amount of RAM. This would require new hardware, which isn't a huge deal, but it's still a fairly large language which we couldn't cover in any detail in our class time.

I finally decided to just try and write a small Python-inspired language that could fit on the Arduino. An Arduino Duemilanova has:

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

In modern terms, that's a really tiny machine. In particular, to avoid having to erase and re-write the Flash constantly, I decided to restrict applications and data to RAM, and to store source code in EEPROM.

Chapter 2. Introducing Snek

The goals of the Snek language are:

- Text-based. Instead of building software using icons and a mouse, a text-based language offers a richer environment for people comfortable with using a keyboard.
- Forward-looking. Skills developed while learning Snek should be transferable to other development environments.
- Small. 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 (version 3) system, but few Python3 programs will run under Snek.

Chapter 3. 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 just provide some a framework within which those details can be filled in later on.

This tutorial shows what appears on the screen, which merges what Snek displays along with user input. User input is shown on the lines which start with **>** or **+**, Snek output is shown on other lines.

3.1. 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 simply evaluate an expression. Start up Snek on your computer (perhaps by finding Snek in your system menu or by typing `snek` at the usual command prompt):

```
Welcome to Snek version v0.8
>
```

At this prompt, the result of any expression typed in will be printed:

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

Here we see that Snek strings can be enclosed in single quotes. They can also be enclosed in double quotes, which can be useful if you want to include single quote marks in them.

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

Stepping up a notch, instead of simply inputting the string directly, we can write an expression which computes the result:

```
> '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 printing the value directly:


```
> 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 decoration, while the command processor prints values in the same format as they would appear in a program.

Finally, let's write a function which prints the value and call it:

```
> 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 a list of arguments. After that list there's a colon.

Colons appear in several places in Snek and they are always 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 long compound statement like this, the command processor will prompt with **+** instead of **>** to let you know that it's still waiting for more input before it does anything. It's the 'line with the old indentation level' that displays the second **+**. Hitting enter on that line ends the list of statements for the **`hello`** function and gets you back to the regular command prompt.

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

So far, in these examples, Snek 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 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 Strings.

3.2. 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 the compiler. 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, rather than just printing it directly.

3.3. 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):
+         print('%f F = %f C' % (F, f_to_c(F)))
+
> 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
```

First off, we've got a new statement, the `for` statement. This walks over a range of values, assigning the control variable (F, in this case) all of the values in the range and then evaluating the statements within it. The range operator creates this set of values for F by starting at the first value and stepping to just before the second value. Each time, it steps by the third value.

You can elide the first value and Snek will use 0 as the starting point. You can elide the third value and Snek will step by 1.

Second, we need to insert the numeric values into the string shown by print. In many languages, that's done with a special formatted printing function. In Snek, there's a more general purpose mechanism called 'String Interpolation'. Using the % operator, Snek walks over the string on the left and inserts values from the set 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 inserted depends on the character following the % mark; that's discussed in the section on String Interpolation below. How to format that set of values on the right is discussed in the next section on Lists and Tuples.

3.4. 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 above, 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 the Snek machine 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 range form above, 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 with the + operator:

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

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

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

3.5. Dictionaries

Dictionaries are the fanciest data structure in Snek. Like Lists and Tuples, Dictionaries hold multiple values. Unlike those, Dictionaries are not indexed by numbers. Instead, Dictionaries are index by another Snek value. The only requirement is that the value be unchanging, which means Dictionaries can only be indexed by immutable values. Lists and Dictionaries are the only mutable data structures in Snek, so there are lots of options for Dictionary indices.

The indexing value in a Dictionary is called the 'key', the indexing 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 the same key, although you're free to specify them with duplicate keys; only the first 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 }
```

This example shows creating the Dictionary and assigning it to d, then fetching elements of the dictionary and assigning to them. You can add elements to a dictionary by using a index which is not already present.

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

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

3.6. While

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

```
> def f_to_c_table():
+     F = 0
+     while F < 100:
+         print('%f F = %f C' % (F, f_to_c(F)))
+         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 above; it first assigns F to 0, then iterates over the statements until F is no longer less than 100.

3.7. If

If statements provide a way of selecting one of many paths of execution. Each block of statements is preceded by an expression, if that expression is 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, lower case 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 lower %d upper %d other %d' % (d, l, u, o))
+
> count_chars('4 Score and 7 Years Ago')
digits 2 lower 13 upper 3 other 5
```

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

Chapter 4. Lexical Structure

Snek programs are broken into a sequence of tokens by the compiler, then the sequence of tokens is recognized by a parser.

4.1. 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 either an integer-part or a fraction and if only the fraction, then that must have at least one digit.

32-bit IEEE floating point values 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 (plus and minus) and one value of NaN. Because Snek does not have an explicit integer type, computations on integer values will convert floats to integers, perform the operation and convert back to floats. Operations on values more than 24 bits wide will lose precision in this process.

Only positive numbers are part of a Snek program; use the unary minus operator to construct negative values.

4.2. 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
```

4.3. Keywords

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

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

4.4. Punctuation

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

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

4.5. 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 are indented the same amount; statements in deeper blocks are indented more, statements in external 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.

4.6. String Constants

String constants in Snek are enclosed by 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.

\t

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

\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.

4.7. List and Tuple Constants

List and Tuple constants in Snek are values separated by commas enclosed in either square brackes (for Lists) or 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.

4.8. Dictionary Constants

Dictionaries in Snek are key/value pairs separated by commas and all 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; 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 5. Data Types

Like Python, Snek does not have type declarations. Instead, each value has an intrinsic representation and all variables may hold values of 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 dispenses with integers and provides only 32-bit IEEE floats. Integer values of less than 24 bits can be represented exactly in these floating point values, so 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. However, 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 an ordered set of values. You can change the contents of a list, add or remove elements. In other languages, these are often called arrays or vectors. Lists are 'mutable' values.

Tuples

Tuples are immutable lists of values. That is, you can't change the contents of a list once created, although if one of the elements of the list **is** mutable, you can modify that 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. However, unlike Lists, 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.

5.1. 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 6. 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 many (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 division on integer values, producing an integer result.

value* % *value

The Modulus operator computes an integer remainder on integer values. 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 below 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 Not operation on its one 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.

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

The Index operator selects the *index* 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.

6.1. 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 begining.

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 begining. 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`. Here are some examples:

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

6.2. String Interpolation

String interpolation in Snek can be confused with formatted printing in other languages. In Snek, the `print` function takes a single S. String interpolation is how this String is generated from a format specification String and a List or Tuple of parameters.

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 which 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 (hexidecimal), 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 interpolatiton is using the first element of **a** as the value instead of using all of **a**.

Chapter 7. 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 left 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 Datatypes chapter.

Chapter 8. Control Flow

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

8.1. `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.

8.2. `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`**, then 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.

8.3. `for` *name* `in` *value* `: block` *[* `else: block` *]*

The **`for`** statement assigns *name* to each of the list of *values* and then executes a block of statements. *Value* can be specified in two different ways, either as a List, Tuple Dictionary or String value, 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 of each separate 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.

8.4. `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.

8.5. 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.

8.6. 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`.

8.7. pass

The `pass` statement is a place-holder that does nothing and can be used anyplace 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 9. Other Statements

9.1. `import` *name*

The Import statement is ignored and is part of Snek so that Snek programs can be run using Python3.

```
> import curses
```

9.2. `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.

9.3. `del` *location*

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

Chapter 10. 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.

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

A **`def`** statement declares (or redeclares) 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 11. 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.

11.1. `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
```

11.2. `print(` *string* `, end='\n')`

Print writes a string to the console followed by the end value (default: `'\n'`).

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

11.3. `sys.stdout.flush()`

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

11.4. `ord(` *string* `)`

Converts the first character in a string to its ASCII value

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

11.5. `chr(` *number* `)`

Converts an ASCII value to a one character string.

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

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

Compute the square root of its numeric argument.

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

Chapter 12. Common System Builtin Functions

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

12.1. `exit(` *value* `)`

Terminate snek and return *value* to the operating system. How that value is intepreted 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.

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

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

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

12.3. `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
```

12.4. `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())
>
```

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

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

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

Appendix A: Arduino Built-in Functions

The Arduino version of 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.

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**.

A.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.

A.2. `listento(` *pin* `)`

Sets the current input pin to *pin*.

A.3. `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**.

A.4. `setleft()`

Turns the current Direction pin **on**.

A.5. `setright()`

Turns the current Direction pin **off**.

A.6. `on()`

Turns the current Power pin **on**.

A.7. `off()`

Turns the current Power pin **off**.

A.8. `onfor(` *seconds* `)`

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

A.9. `read()`

Returns the value of the current Input 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`.

A.10. `stopall()`

Turns all pins off.

The ATmega 328P processor also has a small EEPROM on-chip which can hold 1kB of data. Snek uses that to hold source code. This code is read at boot time, allowing Arduino boards with Snek loaded to run stand-alone. These functions are used by Snekde to get and put programs to the device.

A.11. `eeprom.write()`

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

A.12. `eeprom.show()` / `eeprom.show(1)`

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, anda ^C is sent afterwards. **Snekde** uses this feature to accurately capture the program text when the Get command is invoked.

A.13. `eeprom.load()`

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

A.14. `eeprom.erase()`

Erase the eeprom.

Appendix B: 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 Python3 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.

B.1. `curses.initscr()`

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

B.2. `curses.endwin()`

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

B.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 Python3 curses.

B.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.

B.5. `stdscr.erase()`

Erase the screen.

B.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.

B.7. `stdscr.move(` *row* `,` *column* `)`

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

B.8. `stdscr.refresh()`

Flushes any pending screen updates.

B.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.