

# Toolbox Content

## Python Language

### Assignment

#### Binding names to values

You're using an **assignment** statement when you define or update a [variable](#).

```
# Initialize the count
count = 0
```

The above *assigns* the value `0` to `count`.

- In Python the *variable* `count` is just a "name", and *assignment* is defined as "binding *names* to *values*".

#### Right to Left

An expression like `x = x + 1` might be confusing until you understand that the right-hand side is evaluated *before* the left-hand side of an assignment statement.

Read assignments *right-to-left!* So think of this in 2 steps:

- The `x + 1` creates a new [integer](#) value.
- The new value is assigned to `x`.

#### Assign by Reference

A name that's assigned or "bound" to a value is like a sticky-note.

- The name doesn't *contain* the value, it just *labels* it.
- And just like many sticky-notes can be on the same box, more than one name can refer to the same value.

For example

```
# The names 'items' and 'a' in the following both reference the same List:
items = [5, 6, 7]
a = items
a[0] = 4
print(items) # prints [4, 6, 7]
```

Notice this means you can't *copy* an object like a [list](#) by simple *assignment*. If you wanted a separate copy of the list above you could use `a = items.copy()`, to bind the variable name `a` to a *new list* object returned by the `copy()` method.

#### Augmented Assignment

Expressions like `y = y + 5` are very common, so there is a shorter way to write them!

```
y += 5 # Add 5 to y
```

You can replace the `+` operator above with any of the [binary operators](#):

```
y *= 2 // double the y-value
x -= 1 // subtract 1 from x
```

These forms of assignment can even be more [efficient](#) than their longhand equivalents, since the target of the assignment can be updated in-place rather than first creating a new object prior to assignment.

#### Multiple Targets: *unpacking*

The *left-hand side* of an assignment statement can be a *list* of names!



- If so, the *right-hand side* must be [iterable](#), and the names on the left are bound to the sequence in order.

```
a, b, c = (1, 2, 3) # result: a==1, b==2, c==3
```

This is also known as "unpacking" a sequence into [variables](#).

## Un-Assigning

Can you "unbind" a name once it has been assigned to? Of course you can! This is Python after all :-)

```
direction = "North"
del direction
print(direction) # NameError: name 'direction' is not defined
```

Assignment statements: [https://docs.python.org/3/reference/simple\\_stmts.html#assignment-statements](https://docs.python.org/3/reference/simple_stmts.html#assignment-statements)

# Bitwise Operators

## Manipulating integers at the binary level

Python provides several built-in operators for manipulating [integers](#) at the binary level.

### Shifting Left and Right

As you might expect from the name, these operators shift the bits to the left or right by the specified number of positions. For example:

```
x = 0b0010
y = x >> 1 # Shift right: y = 0b0001
y = x << 1 # Shift Left: y = 0b0100
y = x >> 2 # Shift right: y = 0b0000
y = x << 2 # Shift Left: y = 0b1000
```

### Logical Operations: and, or, xor, not

Bitwise logical operations perform the fundamental [boolean](#) logic functions:

Operation	Symbol	Result for Each Bit	Example: A op B
AND	&	1 if both A and B are 1	0b0110 & 0b1110 => 0b0110
OR		1 if either A or B are 1	0b0110   0b1110 => 0b1110
XOR	^	1 if only A or B is 1	0b0110 ^ 0b1110 => 0b1000
NOT	~	Flip from 1 to 0 or vice-versa	~0b0110 => 0b1001

Bitwise Operations: <https://docs.python.org/3/library/stdtypes.html#bitwise-operations-on-integer-types>

# Blank Lines and Whitespace

## Spaced out code

Like most programming languages, *Python* ignores blank lines in your code.

Adding blank lines can help make your code more *readable*. A good practice is to use blank lines to separate logical sections of your code.

Additional *Whitespace* (spaces) in your code should be used in keeping with the accepted style of the language. The example code in *CodeSpace* follows the official Python Style Guide (PEP-008).

Style Guide: <https://www.python.org/dev/peps/pep-0008/>

---

## ***bool***

### ***Booleans, True or False***

The flow of your code often depends on whether a **condition** is True or False.

**True** and **False** are called **Boolean** values, named for a famous Englishman, *George Boole* - one of the great mathematical geniuses of his day. Back in the mid 1800's he invented the way we express logic in computing today. I think he would have *loved* CodeSpace!

The values `True` and `False` are special *Python* keywords too.

- Your code will often test conditions, like `"Is count > 22 ?"`
- The **result** of this sort of test will be either `True` or `False`!

A **bool** is a value that can be **True** or **False**.

You can define your own **bool** variables:

```
game_over = False
while game_over == False:
    # Continue playing game!
    next_level()
```

In Python, you can convert other types like `string` or `int` to **bool** with `bool()`.

#### **"Truthy" or "Falsy"?**

How does Python decide if a non-bool type is `True` or `False`?

- Zero values and *empty strings or lists* are `False`.
- Other values are `True`.

Ex:

```
bool(0) # is False
bool(1) # is True

bool("") # is False
bool("Hello, World!") # is True
```

**Need an expression that's *always* True ? You could write:**

- `True is True`
- ...But it's easier just to use `True` by itself!

Ex:

```
# The classic infinite loop in Python
while True:
    do_something_forever()
```

**Boolean Values:** <https://docs.python.org/3/library/stdtypes.html#boolean-values>

---

## ***Branching***

### ***Decision points in code***

As your code runs, one line at a time, there will be points where a decision has to be made.



Your code will take a **different branch** depending on the value of  $x$  or some other condition.

- The *condition* is a [Boolean](#) value, often the result of a **comparison operator**.

The `if condition_A` statement tells Python to only run the block of code [indented](#) beneath it if *condition\_A* is `True`.

An `if` statement can be followed by one or more `elif` statements. That's short for "else if", and means the following block will run *only* if the prior `if` or `elif` was `False`.

```
if condition_A:
    # Do something amazing...
elif condition_B:
    # Do this only if condition_B is True
    # (...and condition_A was False!)
```

An `if` statement can end with an `else` statement:

```
if condition_A:
    # Do something amazing...
elif condition_B:
    # Do this only if condition_B is True
    # (...and condition_A was False!)
else:
    # Finally, do this if all of the prior if/elif conditions were False
```

---

## Break and Continue

### Loop control flow statements

Are you're *stuck* in a [loop](#) ?

- Fear not, Python gives you a couple of ways to get *un-stuck*!

#### Break OUT!

A loop will continue until the [condition](#) becomes `False` or the [iterable](#) is exhausted. But what if you want to break out early?

```
# Wait until button 0 is pressed
while True:
    if buttons.is_pressed(0):
        break
```

That's what `break` is for! Use it to break out of the nearest enclosing loop.

#### Roll On!

Normally the code inside your loop runs until the end of the [indented](#) block.

- But sometimes you may want to *skip back to the top* and run the loop condition again.
- That's where `continue` comes in. It jumps right back to the top of the nearest enclosing loop.

break and continue: [https://docs.python.org/3/reference/simple\\_stmts.html#break](https://docs.python.org/3/reference/simple_stmts.html#break)

## Built-In Functions

### Python's built-in functions

Python has a number of [functions](#) built into it that are always available. See the link below for the full list of built-ins. A few common ones are listed here:

```
# Partial List of Python 3 built-in functions:
abs(x) # Return the absolute value of a number.
all(iterable) # Return True if all elements of the iterable are true (or if the iterable is empty).
any(iterable) # Return True if any element of the iterable is true. If the iterable is empty, return False.
bin(i) # Convert an integer number to a binary string prefixed with "0b".
chr(i) # Return a single-character string corresponding to the given integer representation.
hex(x) # Convert an integer number to a lowercase hexadecimal string prefixed with "0x".
input([prompt]) # Read a line from the console and return a string. First display 'prompt' if present.
isinstance(object, type) # Return True if object is an instance of the given type or class.
len() # Return the number of items in a sequence or collection (such as length of a string).
max(iterable)
max(arg1, arg2, ...) # Return the largest item in an iterable or given arguments.
min(iterable)
min(arg1, arg2, ...) # Return the smallest item in an iterable or given arguments.
open(file, mode) # Return a file object opened in the mode specified
ord(c) # Return the integer internal representation of the given character
print(...) # Print given objects to the console.
range(stop)
range(start, stop, step) # Return a sequence of integers in the given range.
reversed(seq) # Return a reverse iterator for the given sequence.
round(number, [ndigits]) # Return a number rounded to ndigits precision after the decimal point. If ndigits is omitted
sum(iterable) # Return the sum of all elements of the given iterable.
type(object) # Return the type of an object, as a string.
```

**Note:** To see a complete *current* list of built-ins for *Firia IoT Python* devices, run this on the [REPL](#):

```
import builtins
dir(builtins)
```

**Built-ins:** <https://docs.python.org/3/library/functions.html#built-in-functions>

## Comments

### Notes to the human readers of your code

**Comments** are a way to put notes in your code that are **ignored** by the computer. In Python, the **#** symbol is used for comments.

- Any text to the *right* of the **#** is *ignored*.

```
# Check to see if we've reached combustion temperature
if temperature > 451:
    extinguish = True # Activate the sprinkler system
```

If comments are *ignored*, then what's the point?

Sometimes it is very obvious *what* a section of code does, but *sometimes* it is not! And very often it is not clear **why** this piece of code exists!

- Comments are notes to your future self, after you've forgotten all about this code.
- Comments are notes to another person who is trying to figure out what this code does.

### Guideline: **Big Scope** → **Big Comment**

If the *scope* of the code is small, then a *small* comment or *none at all* may suffice. Strive to always *comment about the reason* rather than the coding details. And avoid comments that are *doomed* to become **untrue** one day!

- Which of the following comments is more useful to the reader?
- Which one will still be accurate if the line of code is changed to something besides **5**?

```
# Set value to 5.
value = 5

# Start with an initial maximum value.
value = 5
```

If the *scope* is big, like the **whole program/module** or a **whole function**, then a bigger comment may be good.

- What is the purpose of this program? This function?

*Python has a special type of **comment** used to document functions and modules (file-level)*

### Documentation Strings (**docstrings**)

Multiline comments surrounded by triple-quotes are used at the top of a file, or after a function definition, to document **what this code does**.

```
"""This program makes the CodeBot rotate counterclockwise.
   You can start or stop the rotation by pressing BTN-0.
   """

def check_press():
    """Check to see if BTN-0 has been pressed."""
    return buttons.was_pressed(0)
```

There are **Automatic Documentation Generator** tools that can use **docstrings** to create *browsable* documentation for your code!

Read through the section on *comments* in Python's Style Guide. *Proper comments are important!*

**Style Guide:** <https://www.python.org/dev/peps/pep-0008/>

## Comparison Operators

### Testing different conditions

Expressions like `x > 9` and `name == "Guido van Rossum"` let you **compare** two values.

- The result of a *comparison* is a **True** or **False** **boolean** value.

**For Example:** given that the value assigned to `x` is `5` :

```
x > 10 is False
```

```
x < 10 is True
```

```
x == 5 is True
```

Another example: **branching**

```
if distanceToWall < 10:
    motors.run(LEFT, -50)
    motors.run(RIGHT, -50)
```

In the example above, `<` is the "less than" operator, and it makes the expression `distanceToWall < 10` evaluate to `True` if the value of `distanceToWall` is less than 10.

You can read the expression almost like a sentence: "If distanceToWall is *less than* 10, run the code below."

### Comparison Operators:

Operator	Description
>	Greater than
<	Less than
==	Equal to
!=	Not equal to
>=	Greater than or Equal to
<=	Less than or Equal to

Read about other [Operators](#) too!

Comparisons: <https://docs.python.org/3/library/stdtypes.html#comparisons>

---

## Constants

### Unchanging values

**Constants** are names, just like [variables](#), that can represent data in your code.

- Like a **variable**, a **constant** has a *name*, *data-type*, and *value*.
- But *unlike* a variable, a **constant's value should never change**.

Unlike some other programming languages, Python does **not** have a built-in way to declare data "constant".

---

In Python, **constants** are just **variables** that *you promise not to change*.

---

By convention, Python constants are named with **ALL\_CAPITAL\_LETTERS**.

- But remember, the language does not *prevent* code from changing those variables.

Ex: [1]

```
PI = 3.14
```

Naming them with **ALL\_CAPS** is a reminder that those values shouldn't change.

---

1. You should *really* use the [math module](#) for this particular *constant* though!
- 

## Data Types

### str, int, float, and friends

"Data" just means *information* your code works with, like **numbers** or **text**.

Every [variable](#) has a type!

```
zipcode = 35758      # an integer type
city = "Madison"    # a string type
temperature = 98.6  # a float type
hot = True          # a bool type
```

You can use the `type()` builtin function to read a variable's type name.

Check out the builtin types: [str](#), [int](#), [float](#), [bool](#), [None](#)

## Default function parameters

### Values for parameters when you leave off arguments

When you define a [function](#) you can specify **default values** for [parameters](#).

- These are **"optional arguments"** - the caller can omit them!
- If the caller omits arguments, those parameters will get their default values.

```
# A test function
def func(foo, baz=4):
    print(foo, baz)

func(1, 2) # No defaults needed. Displays "1 2"
func(3)   # Use the default 'bar'. Displays "3 4"
```

A similar *looking* but different concept is [keyword arguments](#).

## dictionary

### Dictionary - Python's Mapping type

A **dictionary** is a container that holds *keys* and *values*. It supports fast lookup of a *value* based on the given *key*.

#### Keys and Values

You can use any Python data type as a key as long as it is *immutable*. So for example [strings](#), [ints](#), and [tuples](#) can be **keys**, but not [lists](#) or other dictionaries.

Any object can be a value in a dictionary.

A dictionary **literal** is defined with *curly braces* {} like so:

```
# Create a new dictionary
choice = {'yes': True, 'no': False}

# Lookup the values mapped to 'yes' and 'no'
choice['yes'] # True
choice['no']  # False
```

As with [lists](#), items can be added and retrieved using *square brackets* [key].

```
# Start with an empty dictionary
d = {}

# Add an item
d['zero'] = 0 # {'zero': 0}
d['one'] = 1 # {'zero': 0, 'one': 1}

# Lookup a value
val = d['zero'] # 0
val = d['six'] # Raises KeyError since key is not in d!
```

#### More ways dictionaries can be *modified* and *accessed*

- Consult Python *dictionary* docs (below) for full details!

```
# Create new dict from a list of (key, value) tuples
d = dict([ ('one', 1), ('two', 2), ('three', 3) ]) # {'one': 1, 'two': 2, 'three': 3}

# Return the value for key if key is in the dictionary, else default.
d.get(key [, default]) # if default is omitted this returns None when lookup fails.
```



```
# Remove an item. Raises KeyError if key is not in the dict.
del d['two'] # {'one': 1, 'three': 3}

# Remove an item and return its value (if key isn't in d, return default)
val = d.pop(key [, default])

len(d) # Return number of items in d
key in d # Return True if d has key

# Iterators to use with for loops, etc.
d.items() # Iterator of (key, value) tuples
d.keys() # Iterator of keys
d.values() # Iterator of values
```

Dictionary: <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

## Escape Sequences

### Special Characters

In many programming languages, a character preceded by a backslash is known as an **escape sequence**.

**Escape sequences** allow you to use illegal or special characters in a [string](#).

#### Saving strings!

For example, the string 'You're the best!' is invalid.

Since we're using single quotes to denote a string, an apostrophe in the string will break our syntax!

However, our string can be saved with an **escape sequence**! 'You\*re* the best!'

#### Newlines!

Another **escape sequence** is the **newline**!

In a surprising twist, it creates... *a new line*!

```
print("Here is one line, and it's above...\n\nThis line!!!")

# Output:
# Here is one line, and it's above...
#
# This line!!!

print("Here is one line, and it's above...\n\n\tThis line!!!")

# Output:
# Here is one line
#
#   This line is indented!!!
```

#### Example Escape Sequences:

```
\\ Backslash
\' Single Quote
\" Double Quote
\n New Line
\t Horizontal Tab
```

See also: [strings](#)

**Escape Sequences:** [https://docs.python.org/3/reference/lexical\\_analysis.html#grammar-token-python-grammar-stringscapeseq](https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-python-grammar-stringscapeseq)

# Exception

## Python Exceptions

Python *exceptions* can happen due to errors in your code.

- Your program execution will stop when an exception occurs unless there is a `try` block to catch the error.

There are many different types of exceptions.

- Some of the most common are:
  - `TypeError` - You tried to use an improper `type` in a function.
  - `ValueError` - Generally when a value supplied is out of range for a function.
  - `NameError` - You tried to reference a variable that doesn't exist.
  - `KeyError` - You tried to reference a key in a `dict` that doesn't exist.
- You can even create your own exceptions by inheriting from the base `Exception`!

`raise` is a keyword that lets you manually create an exception.

- This can be done when you detect an "error" OR as another means of `control flow` for your program.

Some other keywords that are used for exception handling include:

Block	Optional	Description
<code>try</code>	False	Stops as soon as an exception is raised!
<code>except</code>	False	Runs if an exception occurred in the <code>try</code> .
<code>else</code>	True	Runs if there was <b>no</b> exception in the <code>try</code> .
<code>finally</code>	True	Runs after all other blocks no matter what.

Here is an example of exception handling:

```
try:
    raise Exception('Go to the except block!')
    print('Never made it here!')
except:
    print('An exception happened!')
else:
    print('Never got here because there was an exception!')
finally:
    print('Always finish with a finally!')
```

Exceptions: <https://docs.python.org/3/tutorial/errors.html#exceptions>

# Files

## File Operations

### Open

Files in your filesystem can be programmatically accessed through the file object:

```
# Create a new file object
f = open(filename, mode)
```

Parameter `mode` determines which set of operations you can perform on the file object.

Mode	Description
<code>r</code>	Read only mode only allows you to read data.

Mode	Description
r+	Opens file for both reading and writing.
rb	Read only mode for binary data.
w	Write only mode creates a new file, or overwrites an existing one.
w+	Opens file for both reading and writing but overwrites existing file.
wb	Write only mode for binary data.
a	Append mode lets you start writing to the end of an existing file, or creates one if it doesn't exist.
x	Opens a file for write only if it doesn't already exist.

## Iterating Lines of a File

The returned file object `f` is also an [iterator](#) when in a readable `mode`.

- Each iteration returns a line sequentially!

Ex:

```
f = open('testfile.txt', 'r')
for line in f:
    # On first iteration print line 1, second line 2, etc...
    print(line)
f.close()
```

## Close

`f.close()` closes an opened file.

It's good practice to *close* an opened file when your program is finished using it.

- This ensures the file contents are written and allows others to use the file.

## The `with` statement - Context Manager

Closing a file is an example of a "cleanup" task. It's easy to forget this step, or accidentally skip over it by returning from a [function](#) or due to an *exception*.

For such cases Python comes to the rescue again! The `with` statement makes sure that the *enter* and *exit* actions are completed for a suitably equipped object. And the `file` object is so equipped - it will `close()` when the `with` block is exited.

Check out this rewrite of the above example. Much nicer using `with`, and the `close()` happens *automatically*!

```
with open('testfile.txt', 'r') as f:
    for line in f:
        # On first iteration print line 1, second line 2, etc...
        print(line)
```

## Read

`f.read(n)` returns a [string](#) with the specific number of characters `n` from the file.

- When `n` is not specified or is negative, the entire file is read and returned.

## Readlines

`f.readlines()` returns a [list](#) containing a [string](#) for each line of the file.

## Readline

`f.readline()` returns the next line from the file as a [string](#).

## Write

`f.write(bytes)` will write text or byte object `bytes` to the file.

- Write will behave differently depending on the `mode` the file was opened with.

If `mode` is `'a'`, the text will be appended to the end of the file.

If mode is `'w'`, the inserted text will overwrite any existing content.

## Writelines

`f.writelines(list)` will write a [list](#) of items to lines of a file.

- Similarly to write, the functionality of `writelines` changes depending on the mode

If mode is `'a'`, the list items will be appended to the end of the file.

If mode is `'w'`, the inserted list items will overwrite any existing content.

## Flush

`f.flush()` ensures the file contents are saved to the filesystem.

- `flush` happens automatically when you `f.close()`, but there may be times you want to keep a file open while still saving the contents to the filesystem.

`open()`: <https://docs.python.org/3/library/functions.html#open>

IO Module: <https://docs.python.org/3/library/io.html>

`os.path`: <https://docs.python.org/3/library/os.path.html>

`os`: <https://docs.python.org/3/library/os.html#file-object-creation>

---

# float

## Floating point number type

A **float** is a *real* number with a decimal point. It can hold a fraction, like:

```
PI = 3.14
body_temp = 98.6
```

In Python, you can convert an [integer](#) or [string](#) to a **float** with `float()`

Examples:

```
float(7) # Returns 7.0
float("2.71828") # Returns 2.71828
```

**Numeric Types**: <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

---

# Functions

## Reusable chunks of code

A function is a named chunk of code you can run anytime just by calling its name!

In other programming languages functions are sometimes called **procedures**. Functions can also be bundled with *objects*, where they're referred to as **methods**. Whatever you call them, they are a good way to package up useful sections of code you can use over and over again!

In Python you can **define** a new function like this:

```
def flashLEDs():
    leds.user(0b11111111)
    sleep(0.5)
    leds.user(0b00000000)
    sleep(0.5)
```

Once that's defined, you can call the function whenever you like:

```
while True:
    flashLEDs()
```

## Parameters and Returns

You can also define [parameters](#) for a function, like the `x` below. And the [return](#) statement lets the function give a result back to the caller.

```
def addTen(x):
    return x + 10

result = addTen(2)
# Now result is 12
```

## Function Objects

Functions are "first class objects" in Python, so they can be passed around, assigned to new [variable](#) names, used as values in [lists](#) or [dictionaries](#), etc.

- The *name* you use when you define a function becomes part of the *function object*, and is added as a [variable](#) in the scope where you defined it. You can assign it to a different variable name later if you like!

# import

## Referencing other code - library modules

`import` statements in Python let you use code from outside your own source file. These external sources are called *modules* in Python. You will also sometimes hear *modules* referred to as *libraries*.

This is a very important feature of the language, as it lets you leverage the work of others. Example *modules* include `botcore`, `time`, and `random`.

`import` statements can have different forms. Throughout this course you have written code like:

```
from botcore import *
import time
from random import randint
```

and you may have wondered at the slight differences in each statement:

- Why do some of the import statements say *from* but others do not?
- Why do some of the import statements have a `*` in them but other statements do not?
- Why is the word `import` sometimes on the right side of the module name instead of the left side?

These variations in the different `import` statements affect two different behaviors:

- **How much** of the *module* is imported
- The exact **naming** of the features you imported

First let's talk about the "how much", using the `random` *module* as an example. The `random` module contains about eight functions, and one possible way to import them all would be:

```
from random import getrandbits
from random import seed
from random import randint
from random import randrange
from random import choice
from random import random
from random import uniform
```

If you only want one or two features, specific individual imports like these are fine. However, if you want most or all of the features from a *module*, you might as well import **all** of them, and there is a handy shortcut to do just that:

```
from random import * # We now have access to everything the random module can do
```

The `*` character in this context is shorthand for "everything". In computer lingo this is referred to as a *wild-card*. The `random` module is an example of a module containing so many features we usually just import them all with one statement, and ignore the ones we don't need.

This brings us to the difference between `from random import *` and `import random`. Both import statements bring in the same **quantity** of features (it imports **all** of them). The difference is in how those features are utilized afterwards. Features imported via `import random` must be referenced using what is called a *fully qualified name*. For example, to set the random *seed* you would have to type:

```
random.seed(123)
```

Features imported via `from random import *` can be referenced without the leading qualifier:

```
seed(123)
```

Which form of import you use is sometimes a matter of personal taste, and sometimes about avoiding naming conflicts. For example, imagine if your program included the following code snippet:

```
def seed(type_of_crop, rows_to_plant, watering_interval):
    # Pretend there is more Python code here
```

Now imagine *also* wanting to use the `seed()` function from the `random()` module. This is an example where you would want to use `import random` so you could do things like:

```
random.seed(123) # make random numbers be consistent for my testing
seed('corn', 124, 'daily') # plant the first crop
```

## Indentation

### Structuring blocks of Python

See the **block** of code below the `while`?

```
while True:
    leds.user(0b11111111)
    sleep(1)
    leds.user(0b00000000)
    sleep(1)

leds.user(0b11000011) # this line will never run!
```

The **indented** code is offset to the right with four spaces (TAB key). This is how the *Python* language organizes blocks of code.

Statements ending in a colon (:), like `while condition:`, always operate on the **block** of code indented just beneath them.

**Be careful and consistent with your indentation!**

- Use the **TAB** key on the keyboard to help with this.
- Make sure every line of code in a *block* is "lined up" properly on the left!

**Note**

- If you've seen other programming languages, you might have noticed they have a *different syntax* to define blocks of code - often {braces} are used. But with *Python* it's all about the indentation!

## Input Function

### Reading text input from the console

The `input()` function is a Python [built-in](#) that lets your program receive text (keyboard) input from the user.

The API for this function is:

```
input([prompt])
```

The [optional argument](#) `prompt` will be printed to the console, then the function will **wait** for the user to type in some text, followed by the `ENTER` key (newline).

- This [function](#) always [returns](#) a [string](#), even if the user entered a *number*!

**Example:** Ask for the user's name, then greet them!

```
name = input("Enter your name: ")
print("Hello", name, ", nice to meet you.")
```

In CodeSpace just open the `≡ Console` menu at the lower right and be sure to click next to the text cursor so your key-presses can be read by `input()`.

**Input function:** <https://docs.python.org/3/library/functions.html#input>

## *int*

### *Integer number type*

An **integer** is a whole number (no fractions) that can be positive, negative, or even *zero*.

You define an **int** using a number *without* a decimal point.

```
num_trombones = 76
```

In Python, you can convert a decimal number or [string](#) to an **integer** with `int()`

Examples:

```
int(7.9) # Returns 7
int("25") # Returns 25
```

Notice that `int()` does **not** *round* up. It just chops off the *fractional* part!

### **Binary and Hexadecimal too!**

By default when you write an *integer literal* Python interprets it as a decimal (base-10) number. But you can also define an integer in [binary](#) (base-2) or **hex** (base-16) by using *prefixes* `0b` or `0x` as shown below:

```
# Set value to 12 (that's 1100 in binary)
value = 0b1100

# Set value to 12 (that's 0C in hexadecimal)
value = 0x0C
```

The above statements produce *exactly* the same result as:

```
# Set value to 12
value = 12
```

**Numeric Types:** <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

# Iterable

**An object capable of returning its members one at a time.**

When you move through a sequence like a [list](#) or a [tuple](#) *one item at a time*, that's called **iterating**.

- Python data types like *lists* and *tuples* are said to be **iterable**.

The [for](#) loop is made to *iterate* over a sequence. This can be one of the sequence types like *lists* or *tuples*, but can also be *any* object that implements the **iterable** interface, such as [strings](#).

- A common example is the [range](#) object. It represents a range of integers, and implements the required functions (methods) to allow code such as *for loops* to iterate over it.

**Iterable:** <https://docs.python.org/3/glossary.html#term-iterable>

---

# Keyword and Positional Arguments

## Passing data to functions with style

When you call a [function](#) the default way that arguments work is based on their *position* in the list when the function is called.

These are called "**Positional Arguments**"

You can also specify your arguments *by name* when you call a function.

These are called "**Keyword Arguments**".

```
# A test function
def func(foo, baz):
    print(foo, baz)

# ALL of these function calls display "0 1" on the Console
func(0, 1)           # Call func with positional arguments
func(foo=0, baz=1)  # Call func with keyword arguments
func(0, baz=1)     # Mix positional and keyword arguments
func(baz=1, foo=0) # Change the order.

# ERROR! No positional arguments after a keyword argument
func(foo=0, 1)
```

A similar *looking* but different concept is [default parameters](#).

**Function Parameters:** <https://docs.python.org/3/glossary.html#term-parameter>

---

# list

## List type

A **list** is a sequence of items that you can access with an *index*.

```
# Define a List of 3 color strings
colors = ["Red", "Green", "Blue"]

colors[0] # "Red"
colors[1] # "Green"
colors[2] # "Blue"
```



```
len(colors) # 3
```

Use *square brackets* `[]` to define a *list*, and to *index* an item.

**Notice:** The *first* item in a list is at *index = 0* !

**Lists can be modified:**

Here are a few ways. Consult Python *sequence* docs (below) for more!

```
# Replace an item:
colors[1] = "Orange" # New colors == ["Red", "Orange", "Blue"]

# Delete an item:
del colors[1] # New colors == ["Red", "Blue"]

# Append an item:
colors.append("Yellow") # New colors == ["Red", "Blue", "Yellow"]

# Insert an item at index i: insert(i, new_item)
colors.insert(0, "Black") # New colors == ["Black", "Red", "Blue", "Yellow"]
```

**More list features:**

There are a lot more capabilities of *sequence types* like lists. Below are a few that you might find useful:

```
# Example List
seq = [100, 101, 102, 103]

seq[i:j] # slice of seq from index i up to but not including j
seq[1:3] # [101, 102]

len(seq) # (4) Number of items in list
min(seq) # (100) Smallest item
max(seq) # (103) Largest item

101 in seq # (True) True if an item of seq is equal to 101

# Make a copy of seq
new_seq = seq.copy()
```

Also see [list comprehension](#) for a compact way to create *lists*.

The *list* is a *Mutable Sequence* (items can be changed) so it has all the *Common Sequence* plus the *Mutable Sequence* operations described in the Python docs link below. Check out [tuples](#) for an *immutable* alternative to lists!

**Sequences:** <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>

## List Comprehension

### Compact and powerful way to build lists

It is common to create a new [list](#) with a [loop](#) that starts with an empty list and appends items like so:

```
even_numbers = []
for x in range(5):
    even_numbers.append(x * 2)

print(even_numbers) # Displays: [0, 2, 4, 6, 8]
```

Python provides a concise way to do the same thing, called a *list comprehension*.

```
even_numbers = [x * 2 for x in range(5)]
```

A *list comprehension* consists of square-brackets containing an expression followed by a `for` clause, and *optionally* an `if` clause. You can even use multiple `for`s and `if`s - see the Python doc link below for more examples.

```
skip_5 = [i for i in range(10) if i != 5] # [0, 1, 2, 3, 4, 6, 7, 8, 9]
```

**List Comprehensions:** <https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions>

## Locals and Globals

### Variable scope and lifetime

Variables that you define outside of a function are called **global** variables.

- Their "lifetime" (how long they retain value) is the whole time the program is running, and their "scope" (where they can be seen/used by code) is the whole file. That's *global*, dude!

The other kind of variable is **local**. Variables created inside **functions** are *local*.

- They only exist while the function is running, then they go away. That's *local* scope.

**What if I want to change a global variable from inside a function?**

- You have to declare it in the function with the `global` keyword, like:

```
count = 0

def check_buttons():
    global count      # <----- Tells Python to use the GLOBAL
                    #                   variable, not to make a LOCAL one.
    if button_b.was_pressed():
        count = count + 1
    return str(count)
```

Now when you assign to `count`, you're assigning to the **global** one rather than creating a new **local** one.

**Note:**

Variables that are *referenced* inside a **function** but **not** assigned a value within it are assumed to be global. So your code can access any global variable it can see! Only when you need to **change** the variable is a `global` declaration required.

## Logical Operators

### and, or, not

You've seen how **branching** and **loops** control the flow of your program with **True / False** decision points such as:

- **Comparison operations** like `x > 51` (which are also **True** or **False**)
- **Functions** like `button_a.is_pressed()` that *return True* or *False*,

What if you have *multiple conditions* to compare - like **two buttons**, testing if *either one or the other or both* is **True**?

```
if buttons.is_pressed(0) or buttons.is_pressed(1):
    print("A button was pressed!")
```

Python provides the following **logical operators** to handle combinations of **Boolean** results:

```
and
or
not
```



`not` is a special kind of logical operator that is only used in its [unary operator](#) form.

```
# Wait for BTN-0 to be pressed
while not buttons.was_pressed():
    pass # Do nothing while waiting.
```

Check out a review of basic [logic operations](#) to better understand the use of these operators.

Read about other [Operators](#) too!

Boolean operations: <https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not>

## Loops

### Repeating sections of code

Ever feel like you're going in circles?

While *people* don't generally like to do the same thing over and over again, *computers* are really great at it!

**Loops** let you change the flow of your code so it **repeats** a block of code again, subject to a [condition](#) you give.

#### while loop

The `while` condition: statement **repeats** the [indented](#) block of code as long as **condition** is `True`.

Example:

This code displays the numbers 0 through 4

```
i = 0
while i < 5:
    print(i)
    i = i + 1
```



#### Breaking Out?

What if you want to break out of a loop before the condition is `True`?

- ...or jump back to the condition check without completing the whole block?
- You can! Check out [break and continue](#) for more on that.

#### for loop

Python's other type of loop is the `for` loop. Combined with the [range](#) iterator it provides a more compact way to achieve the "counting" `while` loop above:

```
for i in range(5):
    print i
```

In general the `for` element `in` iterable: statement **repeats** the [indented](#) block of code once for each **element** of the [iterable](#) (range, string, list, etc.) you provide.

Examples:

This code counts the number of 's' characters in a string.

```
text = "This is a test"
count = 0
for letter in text:
    if letter == 's':
        count = count + 1

print(count)
```

This code displays the numbers 0 through 4

```
for i in range(5):
    print(i)
```

### Target List (*unpacking*)

An additional feature of `for` loops comes from *assignment* [unpacking](#). If you are iterating over a *nested* sequence, as in the following example, the form: `for target_list in iterable:` provides a convenient syntax:

```
tune = [('C', 4), ('D', 2), ('D', 2), ('A', 4)] # note, duration
for note, duration in tune: # Unpack as you iterate!
    play(note)
    sleep(duration)
```

while and for: [https://docs.python.org/3/reference/compound\\_stmts.html#the-while-statement](https://docs.python.org/3/reference/compound_stmts.html#the-while-statement)

## Math Operators

### Math operators, precedence, and math Built-ins

Python has features for doing basic mathematical operations [built-in](#) to the language!

- In addition, there are *many* more math features in the standard [math module](#).

An example of an **operator** is the symbol `*` for **multiplication**.

```
a = 5
b = 2
product = a * b # multiplication operator
```

But if you invite *another operator* to the party, you have to worry about **precedence**.

```
product = 100 + a * b # Hmm... 210 or 110?
```

Consulting the **table below**, you find that `*` is **higher precedence** than `+`.

- So *multiplication binds first*, then *addition*. (product is 110)
- Consider using **parenthesis** to improve *readability* in code like this.
  - Notice which operator in the *table* has the **highest precedence!**

### Operator Precedence Table

The table below lists common Python operators in order from **lowest to highest precedence**.

- Operators toward the top are *lower* precedence (least binding).
- Operators in the same box have the same precedence.

Operator	Description
if - else	Conditional Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
<, <=, >, >=, !=, ==	Comparisons
	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Bitwise Shifts
+, -	Addition and subtraction
*, /, %, //	Multiplication, division, remainder, Integer division
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation (power)

Operator	Description
x[i], x(args)	Subscript, function call
(expressions,...)	Parenthesis

Operator Precedence: <https://docs.python.org/3/reference/expressions.html#operator-precedence>

## Built-in Math Functions

Python has a number of [built-in](#) functions that are available to your code with no module `import` needed. Below are some of the *math oriented* ones:

```
abs(x) # Return absolute value of x.
divmod(a, b) # Divide a/b, return (quotient, remainder).
max(arg1, arg2,...) # Return maximum value of given arguments.
min(arg1, arg2,...) # Return minimum value of given arguments.
round(x [, ndigits]) # Return x rounded to ndigits precision
# after the decimal point.
```

---

## None

### The Null Object

`None` means "no value." It is a special object with *type* `NoneType`, and it is the value returned by [functions](#) which have no [return](#) statement.

`None` is a *falsy* value, and it is often used to initialize variables prior to their being assigned values of a working type.

**None:** <https://docs.python.org/3/library/stdtypes.html#the-null-object>

---

## Parameters, Arguments, and Returns

### Getting things in and out of functions

#### Functions are a two-way street!

- You can pass arguments **IN** to a function when you *call* it.
  - Send arguments to a function inside *parenthesis*.
- When it finishes, it can return things **OUT** - back to your code that *called* it.
  - A function can use the `return` statement to send a *value* back to the caller.
  - The `return` statement ends the function.
  - Your code substitutes the *return value* for the `call()`.
  - If a function has no `return` statement, it returns [None](#) by default.

Here's a simple function that calculates the *square* of the number that's passed to it:

```
def square(n):
    return n * n
```

Now we can call this function whenever we need it:

```
area = square(5)
```

...In this case, **area** will be 25.

The `return` statement:

- *Exits* the function
- Replaces the *calling* statement with the returned value

## Key Terminology

When you **define** a new function, you can declare a list of *names* in parenthesis.

- These are called **parameters**
- Inside the function, they act like *local variables*

When you **call** a function, you can *pass* it a list of **values** in parenthesis.

- These are called [arguments](#)
- They are used to initialize the **parameter** values.

Learn more about [Positional arguments](#) and [Keyword arguments](#)

# Print Function

## Printing text messages to the console

The `print()` function is a Python [built-in](#) that lets your program display text messages to the user. You pass it a series of [positional arguments](#), and they are converted to [strings](#) and printed on the local console interface.

- Objects are *automatically* converted to [strings](#) just like the `str()` function does.
- A single space is printed to *separate* each object printed.
- A newline (`\n`) is printed at the *end* of the print.


You can change the formatting of *separate* and *end* with the `sep` and `end` [keyword arguments](#)

```
print("lute", "bar", sep='+', end='d') # Prints "Lute+bard"
```

## Fancier Formatting

For more control over your text display, check out Python's [string formatting](#) features.

## Getting to the Console

In CodeSpace just open the  **Console** panel at the lower right to see `print()` output. This is also where you can interact with the [REPL](#).

**Print function:** <https://docs.python.org/3/library/functions.html#print>

# Punctuation

## Symbols used in coding

Strange looking *punctuation* can take some getting used to when you first start coding!

For example, these two lines simply display a HEART image, but look at the *asterisk*, *dots*, and *parentheses* - oh my!

```
from codex import *
display.show(pics.HEART)
```

Well, all this *punctuation* has a *purpose*.

- We are using the **codex module** - pre-loaded code that makes it easier to do things with the CodeX.
- The `*` means "import **everything**" from that module (it's called a *wildcard*).
- The `codex` module provides the `display` and `pics` objects.
- When we type: `display.show` we are using the `show` property of the `display` object.

- The `.` dot is a way of accessing part of the display object.
- The **parentheses** `()` tell the computer to **run** a particular piece of code, optionally passing it some data (inside the parentheses).

So, `display.show(pics.HEART)` means

- Run the `display` object's `show` code, passing it the `pics.HEART` data.

## Ranges

### Sequences of numbers you can iterate over

The built-in `range(start, stop, [step])` provides an [iterable](#) object when your code needs a sequence of [integers](#).

At minimum, you have to provide the `stop` argument. If you call `range(n)` with only a single argument, it is assumed to be the `stop` value, with `start=0` and `step=1`.

- **Note:** the range goes up to, but *not* including `stop`.

```
seq = range(5) # seq is (0, 1, 2, 3, 4)
```

You can also include both `start` and `stop` arguments:

```
seq = range(2, 5) # seq is (2, 3, 4)
```

Finally, you can add the `step` argument:

```
seq = range(1, 10, 2) # seq is (1, 3, 5, 7, 9)
```

### Want to count backwards?

Just use a *negative* value for `step`.

Range: <https://docs.python.org/3/library/stdtypes.html#ranges>

## str

### String type

A **string** is a sequence of characters, like words or sentences.

```
name = "Firia"
occupation = "Teaching Robot"
```

Notice that you surround characters in `"` quotes to create **strings**.

Actually, you can use *single* or *double* quotes:

```
name = 'Firia'

# Use different types of quote-marks to enclose quotations:
salutation = 'Just call me "Firia" if you like.'
```

You can convert other [types](#) to *string* with `str()`

Examples:

```
str(7) # Returns "7"
```

**More *string* features:**

There are a lot more capabilities of *string* types. Below are a few that you might find useful:

```
# Example string
text = "This is a test"

text[i] # Return a single-character at position 'i' in the string
# Ex: text[0] is 'T', and text[1] is 'h'

text[i:j] # slice of string from index i up to but not including j
# Ex: text[1:3] is "hi"

text[i:] # slice of string from index i to end of string.
# Ex: text[5:] is "is a test"

text[:j] # slice of string from 0 up to but not including j.
# Ex: text[:4] is "This"

len(text) # (14) Number of characters in string

text = text + "er" # Append another string
# Now text is "This is a tester"

text.split() # Return a List of "words" from the string (separated by spaces)

text.split(',') # Return a List of comma-separated "words" from the string

# Convert between the ASCII value (ord) and Character symbol (chr) of a 1-char string
ord('A') # 65
chr(65) # 'A'
```

See [🔗 Character Encoding](#) for more details on `ord()` and `chr()`

There's much more to learn about **strings** - for example, you can insert special characters in your strings by using [🔗 Escape Sequences](#) which begin with the *backslash* `\` character.

**Escape Sequences:**

Escape Sequence	Meaning
<code>\newline</code>	Backslash and newline ignored
<code>\\</code>	Backslash ( <code>\</code> )
<code>\'</code>	Single quote ( <code>'</code> )
<code>\"</code>	Double quote ( <code>"</code> )
<code>\a</code>	ASCII Bell (BEL)
<code>\b</code>	ASCII Backspace (BS)
<code>\f</code>	ASCII Formfeed (FF)
<code>\n</code>	ASCII Linefeed (LF)
<code>\r</code>	ASCII Carriage Return (CR)
<code>\t</code>	ASCII Tab (TAB)
<code>\v</code>	ASCII Vertical Tab (VT)
<code>\oNN</code>	Character with octal value NN
<code>\xNN</code>	Character with hex value NN

**String:** <https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

**Escape Sequences:** [https://docs.python.org/3/reference/lexical\\_analysis.html#literals](https://docs.python.org/3/reference/lexical_analysis.html#literals)

## String Formatting

### Making strings presentable

Often you have some data that needs to be converted to a [🔗 string](#).



- Maybe you're going to `print()` it to the [console](#), or write it to a [file](#).

The `print` function has some *formatting* capability: passing multiple [arguments](#) combined with the `sep` and `end` parameters. But this can be cumbersome, and doesn't handle things like making decimal numbers print in neat columns.

## Format Strings

You can define a string to serve as a "template" for the string you want to create. This is called a *format string*. Within the *format string* you can put *replacement fields*, designated by curly braces `{ }`.

Example:

```
speed = 25
dist_units = "centimeters"
time_units = "second"
format_str = "Current speed is {} {} per {}."

# Use the format() string method to insert actual arguments into format_str
display_str = format_str.format(speed, dist_units, time_units)

# display_str is now: "Current speed is 25 centimeters per second."
```

## Replacement Fields { }

In the example above the *replacement fields* were empty. In that case the `format()` method maps its arguments into the *format string* in the order that they appear. `format()` can also map [keyword arguments](#) to names placed inside the *replacement fields* like so:

```
output = "Sensor number {s_num} is reading {s_val} degrees.".format(s_val=8.287, s_num=101)
# output is now: "Sensor number 101 is reading 8.287 degrees."
```

If you prefer [positional arguments](#) you can use an [integer](#) inside the *replacement fields* to indicate the order of the `format()` argument to substitute. For example: `"{0} plus {0} equals {1}".format(2, 4)` # "2 plus 2 equals 4"

More details about mapping arguments to *replacement fields* can be found here:

**Replacement Fields:** <https://docs.python.org/3/library/string.html#format-string-syntax>

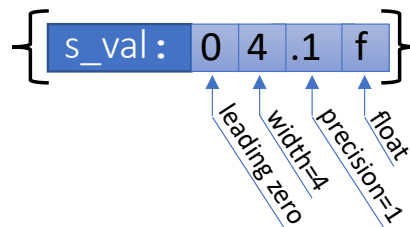
## Format Specifiers - making the format juuuuust right!

When you have **numbers** to display, you can format them *exactly* the way you want to!

- Inside the *replacement field* you can add a colon then a *format specifier*.

```
# Format sensor value: fill with zero's, total width 4 chars, precision 1 decimal place.
output = "Sensor number {s_num} is reading {s_val:04.1f} degrees.".format(s_val=8.287, s_num=101)
# output is now: "Sensor number 101 is reading 08.3 degrees."
```

In the above example the *format specifier* gives a *leading zero*, overall *width* of 4 characters, *precision* of 1 **digit** after the decimal point, with a type of **float**.



The general form of a *format specifier* is:

```
[ [ fill ] align ] [ sign ] [ # ] [ 0 ] [ width ] [ grouping_option ] [ .precision ] [ type ]
```

(Note: Everything in brackets[] is optional... which is everything!)

**Format Specifiers:** <https://docs.python.org/3/library/string.html#format-specification-mini-language>

## f-strings

Huh? Naw, I'm talking about *formatted strings* y'all!

Python provides an even more convenient syntax for string formatting:

```
temp = 67.3253
display_str = f"Temperature is {temp:.1f} degrees"
# display_str is now: "Temperature is 67.3 degrees"
```

Notice the letter 'f' before the quotes? That's an "f-string", and it supports *replacement fields* directly, without the need to call the `.format()` function.

**f-strings:** [https://docs.python.org/3/reference/lexical\\_analysis.html#f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings)

**format() built-in:** <https://docs.python.org/3/library/stdtypes.html#str.format>

# tuple

## Tuple type

A **tuple** is an *immutable* sequence of items that you can access with an *index*.

- The word **immutable** just means the contents can't be changed.
- Think of it as a *read-only* version of a [list](#).

```
# Define tuples of (left_speed, right_speed)
forward = (50, 50)
reverse = (-50, -50)
rotate_right = (35, -35)
rotate_left = (-35, 35)

# Move forward
motors.run(LEFT, forward[0])
motors.run(RIGHT, forward[1])
```

- Use *parenthesis ()* to define a *tuple*.
- Use *square brackets []* to *index* an item.

**Note:** The *first* item in a tuple is at *index = 0* !

### More tuple features:

Tuples support the common capabilities of **sequence types**, like:

```
# Example tuple
seq = (100, 101, 102, 103)

seq[i:j] # slice of seq from index i up to but not including j
seq[1:3] # (101, 102)


len(seq) # (4) Number of items in list
min(seq) # (100) Smallest item
max(seq) # (103) Largest item

101 in seq # (True) True if an item of seq is equal to 101
```

**Sequences:** <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>

# Unary and Binary Operators

## Unary and Binary operators in Python

 **Operators** are keywords and symbols you use to write *expressions* in code. You already know a lot of *operators* from math, for example:

```
2 + 2
```

The plus sign (+) above is the *addition* operator. The numbers on either side are its "operands".

Unary operators only require **one** operand. (*like a unicycle has just one wheel!*)

- Here the '-' symbol is a unary operator:  $x = -1$

Binary operators require **two** operands to work. (*Note: the term "binary" just means "two" in this case. We aren't talking base-2 arithmetic here!*)

- Here the '-' symbol is a binary operator:  $x = x - 1$


In the example below, the '-' operator (unary) is applied before the addition '+' operator (binary).

- Ex.  $x = -1 + 2$

The order that  **operators** are applied follows the  **precedence** rules.

Some operators can be both unary and binary operators.

- For instance, the '-' sign can be used to negate a value (unary) or subtract two values (binary).

The **not**  **logical operator** in Python can only be applied in a unary form.

In Python there are only a few unary operators:

Operator	Description
-	Changes the sign of the value
+	Numeric value is unchanged
~	Bitwise inversion of value
not	Boolean NOT of value

## Underscore

### Making names more readable

A common way to make descriptive names in code is to join words with *underscore*.

On your keyboard, hold down SHIFT and press the dash '-' key.

You'll get used to joining words together like this to make your own unique and meaningful names where needed in your code.

## Variables

### Making up names for things

You might think of **variables** as *boxes* with *labels* on them, that you can put stuff in. That *stuff* might be numbers, text, an Image,... pretty much any of the objects your code needs to work with!

**Ex:** Store the number 73 in a new *variable* called `my_favorite_number`

```
my_favorite_number = 73
```

Now you can use the variable **name** instead of the number. And if you decide to change `my_favorite_number` to something different, you can change it as often as you like! The *variable* lets you store and retrieve *data* to and from the computer's *memory*.

## Naming

Variable *names* in Python have to follow certain rules:

- Must begin with a letter or `_`, not a number
- Can contain letters, numbers, and `_`

### ...but what size is the **Box**?

In some programming languages, defining a variable *creates space* in memory, and you have to specify what [data type](#) a variable will *hold* to be sure there's room. That's very *box-like*.

- But in Python, a variable is really more like a *sticky-note* with a **name** written on it.
- Objects like [integers](#) and [strings](#) can exist without a name, or they can have *multiple names* stuck to them!
- See [assignment](#) for more details about that!




---

## Python Libraries

### Math Module

#### A scientific calculator for your code

Sometimes your *coding challenges* require a little **mathematics**.

Python's **math** module includes a set of mathematical [functions](#) and [constants](#).

- Trigonometric operations
- Logarithms and exponents
- ...and more!

```
import math

# Show off some 'math' module features.
print("pi = ", math.pi)
print("base of the natural logarithm: e = ", math.e)
print("sin(pi/2) = ", math.sin(math.pi / 2))
```



**math:** <https://docs.python.org/3/library/math.html#module-math>

---

## Random Numbers

### Making code unpredictable

Normally a computer can be relied on to be very **predictable**.

Each time you run a program it goes through the same sequence, starting from the first line of code and taking one *predictable* step at a time.

But some applications need **randomness**, or *unpredictable* results:

- Games, where there shouldn't be an obvious *pattern* for the human player to learn.
- Cryptography, where randomness helps secure stored passwords and messages.
- Scientific studies, where *statistical sampling* requires random selection.

On some devices there is a hardware-based "true random number generator" available. When that is not present, Python uses a *pseudo random number generator*, which means the "random" numbers it provides are really just a fixed sequence that's meant to have an *unpredictable* pattern.

To use these functions, they must first be imported from the `random` module:

```
import random
```

If you want to set the starting point in that pseudo-random sequence, use the function `random.seed()`:

```
# Seed the random number generator
random.seed(1234)
```

After *seeding* the random number generator, the following code will always start at the same point in the sequence of "random" numbers.

To generate a random number, use `randrange` or one of the other awesome functions in the `random` module:

```
# Returns a random integer from 0-9
random.randrange(10)

# You can also specify start and stop for the range: [start, stop)
random.randrange(start, stop)

# Return the next random floating point number in the range [0.0, 1.0)
random.random()

# Got a list of items to choose from?
# Return a random item from the given sequence.
random.choice(my_sequence)
```

**random:** <https://docs.python.org/3/library/random.html#module-random>

## Time Module

### Waiting, measuring, and watching the clock tick by.

Python's **time** module provides various time-related functions.

Note that many *embedded systems* like CodeBot and CodeX don't have a "Real Time Clock" enabled, so some of the functions dealing with "time of day" are not available.

- Full implementations of the **time** module have many more features than the MicroPython version. See the links below for details.

For your *embedded MicroPython* applications, the most useful *time* functions are:

- Waiting for specified *delays* with `sleep_XX()` functions.
  - These functions *block* - that is, they don't return until the specified delay has elapsed.
- Checking the current *elapsed time* with `time.time()` or `ticks_XX()` functions.
  - These functions return *immediately* with the current elapsed time count.

#### Note:

- The `time.time()` function is standard Python, available on all platforms. It returns a running count of *seconds*.
  - For systems that have access to a "Real Time Clock" the value is the number of seconds since "the Epoch" January 1, 1970.
- The `ticks()` functions in MicroPython are very useful for marking the passage of time in embedded code.
  - These counters start at **zero** when your device *boots*, and keep counting *up* from there while it's running.
  - **BUT** they can't keep counting to *infinity*! At some point the counters will **wrap-around** back to **zero**.
  - If you're doing calculations based on **ticks** your code needs to account for that! That's where `ticks_add()` and `ticks_diff()` come in handy.



```
import time

# Delay functions: block program execution for specified time interval.
time.sleep(seconds)
time.sleep_ms(milliseconds)
time.sleep_us(microseconds)

# Counter functions: return instantaneous time counts.
time.ticks_ms() # millisecond counter
time.ticks_us() # microsecond counter

# Math that compensates for ticks "wrap-around-to-zero"
time.ticks_add(ticks, delta) # returns (ticks + delta)
time.ticks_diff(ticks1, ticks2) # returns (ticks1 - ticks2)
```

time: <https://docs.python.org/3/library/time.html>

## Timing

### Controlling the pace of actions

Usually a computer program runs as fast as it can, but sometimes you want to slow things down. To use delay functions, they must first be imported from a Python module:

```
from time import sleep
```

The function `sleep()` should only be used when there is no code you want to run during the pause. However, [peripherals](#) like the LEDs and LCD Display will continue to operate while your program "sleeps".

- Functions like this are referred to as *blocking functions*, and calling `sleep()` directly is an example of a *blocking delay*.

```
# Sleep for the given number of seconds
sleep(sec)
```

As an example of how `sleep()` can be used to control the pacing of your program:

```
# Dramatic build up...
display.print("3")
sleep(1.0)
display.print("2")
sleep(1.0)
display.print("1")
sleep(1.0)
# Do something impressive here...
```

## CodeBot Libraries

### Accelerometer

#### Sensing orientation and impact.

Just like a game controller or phone that can sense tilting or shaking, this sensor chip lets the robot detect motion, impacts, and orientation.

An **accelerometer** is a device that measures *proper acceleration*. One way you commonly experience *acceleration* is when you're moving and your speed or direction of motion changes. Think of speeding up in a car, the force you feel pushing you back in your seat or sideways when going through a turn. The forces you're feeling are due to *acceleration*.

So if you're standing still, acceleration is zero, right?

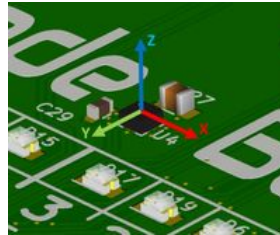
**No!** Well, not unless you're floating in space!

*Gravity* also causes acceleration. So even if your accelerometer is sitting still on your desk, it will measure an acceleration due to the force of gravity pulling straight down toward the floor.

In a *zero-g* environment, like outer space, or if an object is in *free-fall*, the accelerometer will measure *zero* acceleration in all directions.

CodeBot and CodeX have an accelerometer that measures the force of acceleration in 3-directions:

In the picture below, if CodeBot's circuit board is sitting horizontally and motionless on Earth, it will measure *gravitational acceleration* (-1g) in the **Z** direction, but *zero* in the **X** and **Y** directions.



botcore Accelerometer docs: <https://docs.firialabs.com/codebot/kxtj3.html>

codex Accelerometer docs: <https://docs.firialabs.com/codex/lis2dh.html>

## Buttons

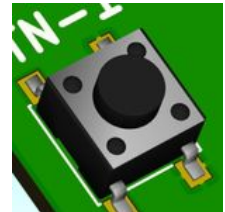
### User interface push buttons 0 and 1

There are two *momentary push buttons* you can **read** from your Python code.

- The buttons are labeled like *bits* of a binary number, **0** and **1**.

The **botcore** library provides functions to read *buttons*:

- Your code can check the **current state** of the buttons with the `buttons.is_pressed(n)` function.
- The library also monitors button presses so your code can check if a button `buttons.was_pressed(n)` since the last time you checked.
  - The **parameter** `n` is `0` for **BTN-0** and `1` for **BTN-1**.



botcore Button docs: <https://docs.firialabs.com/codebot/botcore.html#botcore.Buttons>

## CodeBot LEDs

### Lighting up CodeBot

## Line Sensors

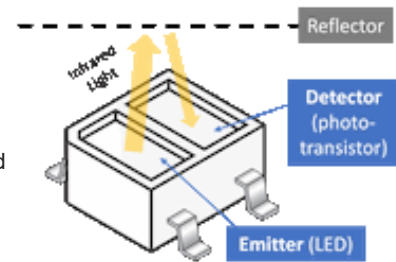
### Detecting lines and boundaries beneath your 'bot

Under CodeBot's front edge are 5 *line sensors*. These little black boxes are known as *photo reflective sensors*.

- They have an **infrared LED** that *emits* light, and
- A **phototransistor** that *detects* light.

Can you see the two sections in each of these sensors?

Your Python code can use these sensors to detect *how much light is reflected* by the surface your 'bot is on. This gives CodeBot the ability to follow lines, detect boundaries, and more!



Example  API usage:

```
# Read a single Line sensor
val = ls.read(2) # Returns analog reading of sensor 2

# Read all Line sensors
thresh = 2500 # Threshold between 'line' and 'ground' on this surface.
is_reflective = False # Black lines
vals = ls.check(thresh, is_reflective) # Returns a tuple of bools
```

botcore LineSensor docs: <https://docs.firialabs.com/codebot/botcore.html#botcore.LineSensors>

## Motors

*Electric motors to power your 'bots wheels*

## Proximity Sensors

*Detecting objects with infrared light*

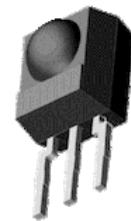
Dual Proximity Sensors give CodeBot the ability to detect nearby objects so you can write code to:


- Avoid obstacles
- Pursue moving targets
- Detect walls

These IR (infrared) sensors can also be used to wirelessly communicate between teams of CodeBots. Let the swarm begin!

The IR emitter is like a very bright "Headlight" for CodeBot, that lights up objects in front of it.

- It emits light in the **infrared** spectrum, which is *invisible to humans*.
- CodeBot uses it together with the *Proximity Sensors* to detect objects based on reflected IR light.



The example code below continuously reads the *proximity* sensors and displays the (boo1, boo1) results on the *prox*  CodeBot LEDs.

```
while True:
    p = prox.detect()
    leds.prox(p)
```

botcore Proximity docs: <https://docs.firialabs.com/codebot/botcore.html#botcore.Proximity>



## Speaker

### Audio output for alert tones, music, and more.

CodeBot's *speaker* can be programmed to play simple audio frequency **itches**, or controlled in much more sophisticated ways through onboard *Digital to Analog Converter (DAC)* or *Pulse Width Modulation* hardware.

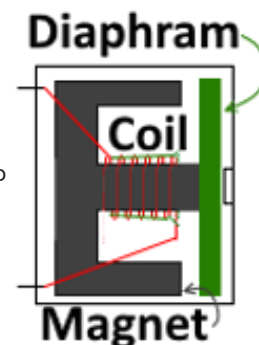
#### I'm a real *speaker* - not a "buzzer"!

Your code sends electric current through the **Coil** causing the **Diaphragm** to move, which vibrates the air to produce **sound**!

- Just like *headphones* or *loudspeakers* in a sound system!

The example code below plays a "2-tone beep":

```
from botcore import spkr
from time import sleep
spkr.pitch(500) # Start playing 500 Hz tone.
sleep(0.1)
spkr.pitch(1000) # Change to 1000 Hz tone.
sleep(0.1)
spkr.off() # Stop playing sound.
```



botcore Speaker docs: <https://docs.firialabs.com/codebot/botcore.html#botcore.Speaker>

## System Status Monitors

### CodeBot system sensors and status reporting

Robots need to be aware of their **internal** environment, in addition to having sensors for *external* stuff!

CodeBot has sensors for:

- CPU Temperature (`temp_C()` and `temp_F()` [functions](#))
- Power supply input voltage
- Power switch position (USB / Battery)

**Example:** A function that lights the **BATT** LED when the battery is low.

```
def check_batt():
    if not system.pwr_is_usb():
        # Read batt voltage under load
        leds.user(0x0f)
        v = system.pwr_volts()
        leds.user(0x00)

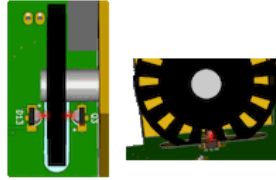
        # Below 25% capacity, turn on BATT LED!
        batt_low = v < 4.5
        leds.pwr(batt_low)
```

botcore System docs: <https://docs.firialabs.com/codebot/botcore.html#botcore.System>

# Wheel Encoders

## Sensing rotation

Your code can control the *power* applied to the motors, but to know exactly how far the wheels have turned you'll need to *sense rotation*. That's the job of the **Wheel Encoders**!



As the encoder disc rotates, an invisible IR (infrared) light beam passes through its slots. Your code can count the **pulses** of light to see how much the wheel has rotated.

**Example:** Show LEFT *encoder* value on the Debug Console.

```
from botcore import *
from time import sleep

while True:
    val = enc.read(LEFT)
    print(val)
    sleep(0.5)
```

botcore Encoder docs: <https://docs.firialabs.com/codebot/botcore.html#botcore.Encoders>

## CodeX Libraries

### Audio

#### CodeX Audio Interface

CodeX has audio hardware that lets you record and play sounds and music.

#### Digital Audio

At the most basic level, computers deal in *numbers*. [Binary](#) numbers to be specific...

- So how do you get sound from numbers?

#### CodeX Ears - the microphone

- Our ears sense analog variations in sound, so this is similar to [Analog to Digital Conversion](#) (ADC) which converts analog inputs to numbers.
  - If you do the ADC repeatedly, really fast, you can "*sample*" a sound wave as a list of numbers.
  - That's how the CodeX **microphone** input works - it's basically an ADC!




#### CodeX Voice - the speaker

- Sound *output* is just the inverse of *input*!
- Instead of an [ADC](#) you need a **DAC** (Digital to Analog Converter).
  - Numbers go in... analog voltage levels come out.
  - Stream a bunch of output voltages to a speaker, and you've got *sound*!


### CODEC - *bringing it together!*

On the back of CodeX right next to the headphone jack is a chip called a **Codec**.

- That chip contains an ADC to *encode* analog sound from the microphone into digital form.
- *And* a DAC to *decode* digital sound from the  CPU into analog voltages for speakers.
- A device that combines a "coder/decoder" is called a **Codec**.

## Example Audio Functions

The audio functions of the CodeX are available in the `audio` object from the `codex` module.

Try playing an MP3 file from the  CodeX Sounds collection:

```
from codex import *
audio.set_volume(65)
audio.mp3('sounds/welcome')
```

Or **record** your voice and play it back!

```
from codex import *
audio.initialize()

# Countdown to record
audio.mp3("sounds/three")
audio.mp3("sounds/two")
audio.mp3("sounds/one")

# Speak into the microphone
buf = audio.record() # Samples for 2 seconds by default
audio.playbuf(buf) # Playback the sampled sound!
```

Check the audio docs link below to learn about other `audio` functions you can use.

The  **soundlib** module provides a higher-level interface for sound effects and tones.


codex audio docs: <https://docs.firialabs.com/codex/codec.html>

## Bitmap

### Graphics bits - drawing images and text

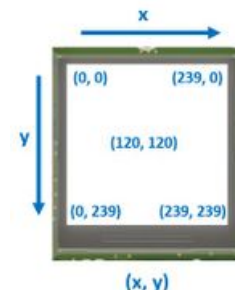
A *bitmap* is an object that can hold a 2D image of a given width x height.

- The image is stored in memory as a list of *pixel*  RGB Color values.
- In addition to providing memory for storing the image, the CodeX `Bitmap` object has functions for drawing graphics and text.

The CodeX `display` object is a `Bitmap` that maps directly to the 240x240 pixel .

- The (x, y) coordinates of the display start from zero at the top-left of the screen.

```
# A few bitmap functions - see full docs link below for more.
set_pixel(x, y, color) # Set a single pixel to specified color
get_pixel(x, y) # Get the color of the given pixel
draw_line(x1, y1, x2, y2, color) # Draw a line from (x1,y1) to (x2,y2)
draw_circle(x, y, radius, color) # Draw a circle outline
fill_circle(x, y, radius, color) # Draw a filled circle
draw_rect(x1, y1, width, height, color) # Draw a rectangle outline
fill_rect(x1, y1, width, height, color) # Draw a filled rectangle
draw_text(text, x, y, color, scale, background) # Draw text, with no wrapping or scrolling
```



codex Bitmap docs: <https://docs.firialabs.com/codex/bitmap.html>

# CodeX Buttons

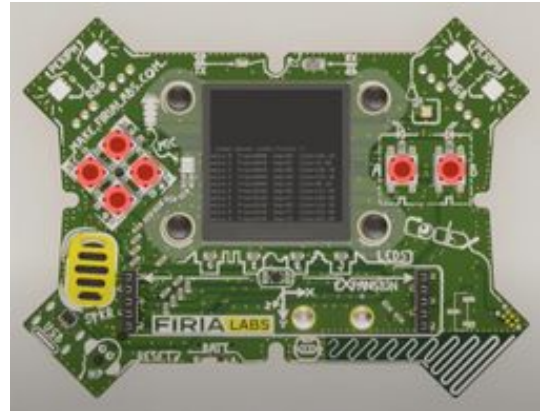
## User interface push buttons

There are six *momentary push buttons* you can **read** from your Python code.

Symbol	Name
BTN_U	Up
BTN_D	Down
BTN_L	Left
BTN_R	Right
BTN_A	A
BTN_B	B

The **codex** library provides functions to read *buttons*:

- Your code can check the **current state** of the buttons with the `buttons.is_pressed(n)` function.
- The library also monitors button presses so your code can check if a button `buttons.was_pressed(n)` since the last time you checked.
  - The [parameter](#) `n` is the symbol from the above table.



**codex Button docs:** <https://docs.firialabs.com/codex/codex.html#codex.Buttons>

# CodeX Image Pics

## CodeX picture gallery

The CodeX can show images known as [Bitmaps](#) on its [LCD](#) screen using the [display](#) functions.

- To keep it simple AND allow you to create your *own* images using "ASCII art", these are retro 8-bit video game style pictures.
- Of course the CodeX can display realistic hi-rez graphics too...

To use **Images**, first import the `codex` module:

```
from codex import *
```

Here are **ALL** the CodeX's pre-defined images:

- `pics.HEART`
- `pics.HEART_SMALL`
- `pics.MUSIC`
- `pics.HAPPY`
- `pics.SAD`
- `pics.SURPRISED`
- `pics.ASLEEP`
- `pics.TARGET`
- `pics.TSHIRT`
- `pics.PLANE`
- `pics.HOUSE`
- `pics.TIARA`
- `pics.ARROW_N`
- `pics.ARROW_NE`
- `pics.ARROW_E`
- `pics.ARROW_SE`
- `pics.ARROW_S`
- `pics.ARROW_SW`
- `pics.ARROW_W`
- `pics.ARROW_NW`

You can also access all the **arrows** in a [list](#) using:

- `pics.ALL_ARROWS`

**Example:** Show an airplane.

```
from codex import *
display.show(pics.PLANE)
```

Check the **codex** docs to learn about other [Bitmap](#) functions you can use.

**codex Bitmap docs:** <https://docs.firialabs.com/codex/bitmap.html>

## CodeX Sound Collection

### Sample sounds

CodeX comes pre-loaded with a few sounds you can use in your programs.

- These are loaded on the *file system*, so you can use your computer to browse for them under the "sounds/" directory on the CodeX itself.
- You can also remove them to free up space, or add new ones by booting CodeX in [USB-writable mode](#).
- Use the CodeX [audio](#) interface to play these sounds.
- Or, for more advanced control, use [soundlib](#).

### Alphabetical listing

Decimal numbers 0-9, button labels, plus a few songs and surprises await you in the built-in sound collection!

CodeX Sounds sounds/...			
a.mp3	eight.mp3	off.mp3	six.mp3
africa.mp3	five.mp3	okay.mp3	techstyle.mp3
b.mp3	four.mp3	on.mp3	ten.mp3
bohemia.mp3	funk.mp3	one.mp3	three.mp3
button.mp3	led.mp3	power.mp3	two.mp3
codetrek.mp3	left.mp3	right.mp3	up.mp3
codex.mp3	mic.mp3	roll.mp3	welcome.mp3
display.mp3	nine.mp3	seven.mp3	yes.mp3
down.mp3	no.mp3	shire.mp3	zero.mp3

**Example:** Play welcome message.

```
from codex import *
audio.mp3('sounds/welcome')
```

## Display

### CodeX display canvas

The CodeX display is an [LCD](#) that allows your code to display full color *text* and *graphics*.

*The display will continue to show the last thing your code sent it, even **after** your program ends.*

To use the display, import the `codex` module:

```
from codex import *


# Display an image (ex: HEART)
display.show(pics.HEART)
```


See the [CodeX Image](#) Gallery for more built-in pics.



CodeX uses a TFT-LCD designed for *smart watch* applications - it's capable of *amazing* graphics!

## The Display API

There are a few Python  modules your code can use to work with the display:

-  **Bitmap** - the `display` object has *Bitmap* functions for drawing text and graphics to the screen.
- **TFT** - use the `tft` object (from the `codex` module) to control `tft.brightness` and `tft.auto_update` behavior.
- **Canvas** - the `display` object is the bottom-layer or "root" Canvas, atop which you can layer other Canvas drawings.

Bitmap docs: <https://docs.firialabs.com/codex/bitmap.html>


TFT docs: <https://docs.firialabs.com/codex/tft.html>

Canvas docs: <https://docs.firialabs.com/codex/canvas.html>

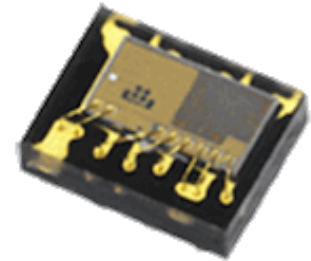
# Light Sensor

## Detect visible and infrared light

The CodeX *light sensor* is a sensitive electronic device which is designed to accurately measure the amount of "ambient light" falling on it. It can measure light in both the *infrared* and *visible* wavelengths of the spectrum.

This sensor converts light intensity to a digital output signal that the CodeX  CPU can read over its I2C interface. It provides a linear response over a wide dynamic range from 0.01 lux to 64k lux.

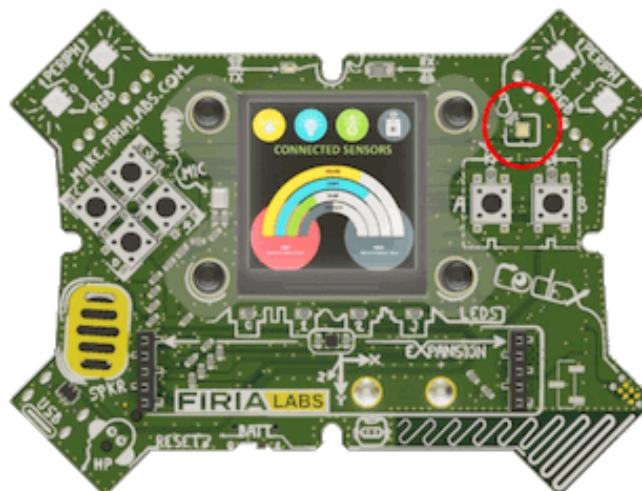
- **lux** is the unit of *illuminance*, defined as the amount of light that a candle from 1m distance would cast on a 1m x 1m square surface.



```
# Example for Light Sensor
from codex import *

# Read raw ADC value from sensor (default max sensitivity)
val = light.read()
display.print(val) # integer: 0 - 65535

# Read value in standard Lux units
lux = light.read_lux(9500) # 9500 Lux range
display.print(lux) # float: Lux
```




CodeX light sensor docs: [https://docs.firialabs.com/\\_modules/codex.html#AmbientLight](https://docs.firialabs.com/_modules/codex.html#AmbientLight)

---

## RGB "pixel" LEDs

### Addressable Multi-Color LEDs

Normal "discrete"  LEDs are simple electronic components that light up one color when an electric current passes through them.

- But what if you want multiple colors?
- And maybe a *string* of LEDs that can be *independently* controlled by your CPU?

### Smart, Addressable LEDs aka "NeoPixels"

CodeX has four of these smart LEDs.

- Each of them has three LEDs: RED, GREEN, and BLUE
- ...plus a tiny *controller chip*!

The *controller chip* allows the CodeX  CPU to send RGB information to it based on its position in a strip (0, 1, 2,...).

You can use any of the standard  RGB Colors with these LEDs, or define your own!

```
# Example usage of CodeX Pixel LEDs
from codex import *
pixels.fill(GREEN) # Set all to GREEN
pixels.off() # ALL pixels OFF
pixels.set(0, RED) # Set pixel 0 to RED
pixels.set([RED, GREEN, BLUE, WHITE]) # Set all 4 to different colors
```

CodeX pixel LED docs: <https://docs.firialabs.com/codex/codex.html#codex.NEOPixels>




---

## RGB Colors

### Custom blended colors

#### Digital Colors: (RED, GREEN, BLUE)

How do computers represent *colors*?

- With *numbers* of course!
- A common way is to use 3 numbers that control the amount of RED, GREEN, and BLUE in a color.

In Python you can use a  list or  tuple to represent an RGB color like this:

```
teal = (0, 255, 128)
pixels.set(0, teal)
```

That's how the CodeX  pixel LED and  bitmap colors are defined. *Feel free to define your own colors!*

When you do `from codex import *` you get the standard color definitions from the `colors` module.

**colors module:** <https://docs.firialabs.com/codex/colors.html>




---

## soundlib

## Sound effects and Tones

The `soundlib` module provides a **high-level** sound generation [API](#).

What's up with *high-level* versus *low-level*?

- At the hardware level the CodeX [CPU](#) sends audio data directly to the CODEC chip, which converts it from *digital* to [analog](#) output on the speaker or headphones.
- The [Codec](#) object that you access with `audio` from the `codex` module provides direct **low-level** sound output features.

The **high-level** functions in `soundlib` give you more capabilities to *create* and *mix* sounds before they're sent to the CODEC.

```
# Example: MP3 running in background with start/stop control
from soundlib import *
funky = soundmaker.get_mp3('sounds/funk', play=False)
funky.play()
# ... do some stuff
funky.stop()
```

**soundlib docs:** <https://docs.firialabs.com/codex/soundlib.html>

## Computer Science

### Algorithm

#### Step-by-step instructions to achieve a goal

You might think of a math problem or a set of instructions to brew a potion when you hear the word **algorithm**.

- Well, you are on the right track!

An algorithm is a set of step-by-step instructions.

- It can be written as text, drawn as a diagram, or coded in a programming language!
- Think about your *algorithm* before and during the process of writing code.
- A good algorithm helps break the program down into logical pieces so that it can be created one step at a time.

Benefits of writing an algorithm *before* you start coding:

- Quick to write
- Simple to read and understand
- Breaks hard problems down into smaller problems

An algorithm can help you [divide and conquer](#) tough concepts.

- A good practice is to make some or all of your steps unique, independent [functions](#)
  - This can also help the [readability](#) of your code

## Analog to Digital Conversion

### You Live in an Analog World

From complete darkness .... to bright sunlight.

From the coldest glacier .... to the hottest desert.

"**Analog**" means **infinite variation** from *dark to light, cold to hot*, and so on.

But what if you want to measure something like **temperature** with a computer?



- A digital computer can't handle an *infinite* number of temperature levels.
- So it converts **analog** measurements to just a few **digits**

For example:

- The CodeX [light sensor](#) converts an *analog* sensor input into a **number** from **0** to **65,535**.

**Aw! Instead of *infinite* brightness variation, we get just 65,536 levels!?!**

Why **65,536**? The computer deals in [binary](#) numbers, and this sensor has a 16-bit ADC:  $2^{16} = 65,536$ .



Fortunately the digital *approximation* of analog measurements is perfectly fine for many applications, like sensing light or temperature with the CodeX.

Think about the online **video** and **music** performances you've seen. They all started as **analog** and were converted to **digital** so we could process, store, and distribute them using computers and **code**!

## API

### Application Programming Interface

This is a TLA (Three Letter Abbreviation) that you will use a *lot* as a software developer!

- The term API (Application Programming Interface) refers to the details of how your program *interacts* with different services it needs.

#### What kind of services?

One example is Python library [modules](#).

- The API of the `import time` module defines the `sleep(sec)` function, and all the other functions it provides.
- There are library APIs to access **LEDs**, **buttons**, and more.
- *You have already been using an API!*

#### Other examples of APIs

Whenever two different programs communicate, the messages they send back and forth are defined by one or more APIs

- For example, a *web browser* and a *web server (site)* have to agree on the API to use for web pages and other services.
- There are APIs that define how program libraries interface with the OS of your computer, to do things like reading and writing files, keyboard input, and display.

Some APIs are published as standards (like the *web* example above) so programs worldwide can interoperate. But there are many more small APIs that are written by software developers every day for every purpose imaginable. As you write code and define [functions](#), [parameters](#), and objects, you will be creating APIs of your own!

## Basic Logic Operations

### Boolean and, or, not

How can [boolean](#) values and [logical operators](#) be used to solve complex logic problems?

#### Basic Operations

Here's a brief description of the basic logical operators: `and`, `or`, and `not`, along with a truth-table for each.

- The `and` operator returns `true` only if both of its operands are `true`. Otherwise, it returns `false`.
- The `or` operator returns `true` if at least one of its operands is `true`. If both operands are `false`, it returns `false`.
- The `not` operator inverts the truth value of its operand. If the operand is `true`, it returns `false`, and vice versa.

**Truth Table:**

A	B	A and B	A or B	not A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

In the table:

- T stands for `true`
- F stands for `false`

## Binary Numbers

### How computers deal with digits

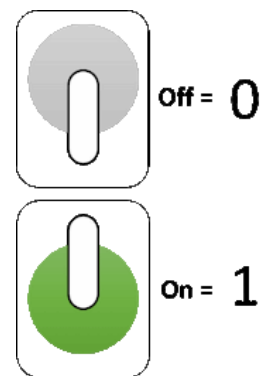
Inside the computer, all the tiny electrical connections that store information are like the **light switches** shown here.

- Each connection can be **ON** or **OFF**
- That's just 2 states!
- The "Bi-" in "Binary" means **2**, just like a "Bicycle" has 2 wheels!

Say you only have **one switch**, and you want to use it to show a number to someone.

- One switch can show one *binary digit*.
- So you can show **2 possible numbers**: **0** (off), or **1** (on)

(This is why it's called a **DIGITAL** computer!)



### What if you have 2 switches?

- There are **4 combinations!**
- Each *switch* is like a *binary digit*
  - It's called a **bit** for short :-)

Look at the switches to the right. See how all 4 combinations are used to make the numbers 0 through 3?

- It's like a **secret code!**
- How big a number could you show with 8 switches? 10 switches? See below...

### The powers of 2

Binary means "base 2", so here's how you calculate the numbers you can make with a given set of switches (bits):

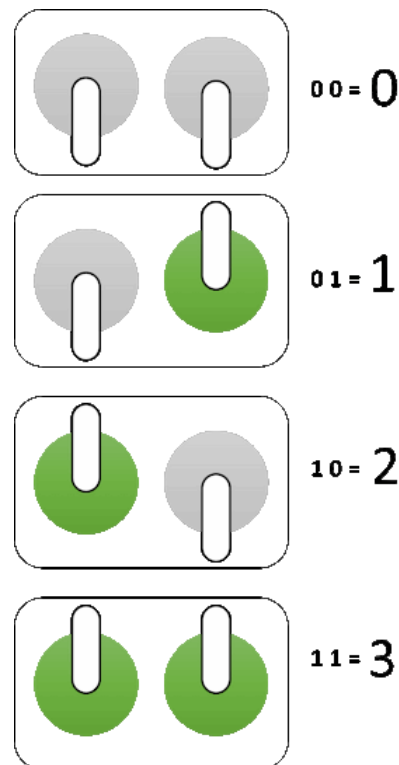
- 1 bit →  $2^1 = 2$  numbers
- 2 bits →  $2^2 = 4$  numbers
- 8 bits →  $2^8 = 256$  numbers
- 10 bits →  $2^{10} = 1024$  numbers

You might have heard of a [byte](#). That's just another name for an **8-bit** number!

### Binary in Python

Usually your code deals with numbers in **decimal**, even though the computer stores them internally in binary form. But Python does have a way to write [integer](#) values directly in *binary* by prefixing the number with **0b**:

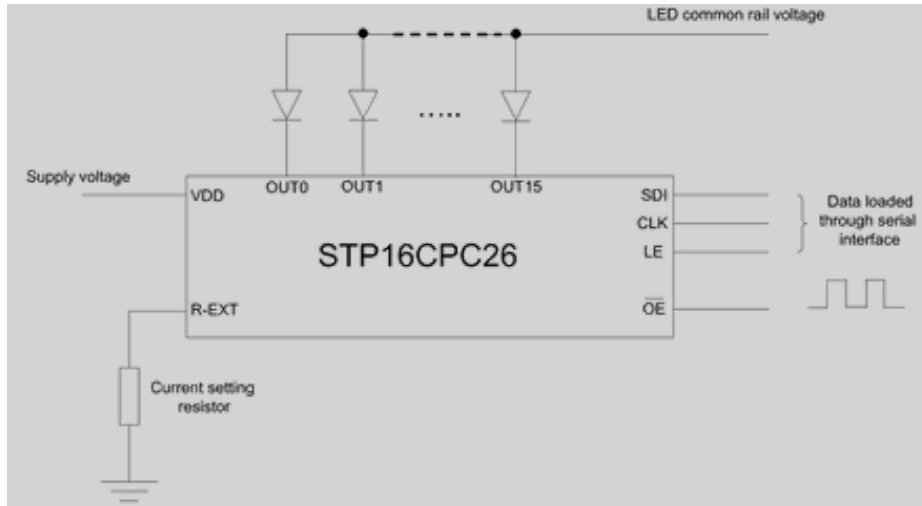
```
# Set value to 9 (that's 1001 in binary!)
value = 0b1001
```



## Binary Shift Register

### A hardware shift register.

A shift register is an electronic circuit that allows an array of output pins to be set HIGH or LOW based on a sequence of binary 1's and 0's that are "shifted-into" a single input pin.



Shift registers are a fundamental building block of computers, and a handy way to control lots of output devices (like LEDs) with just a handful of output pins.

binary shift register: [https://en.wikipedia.org/wiki/Shift\\_register](https://en.wikipedia.org/wiki/Shift_register)

## Byte

### A unit of computer memory.

A **byte** is made up of 8 bits.

With 8 bits you can represent  $2^8 = 256$  numbers.

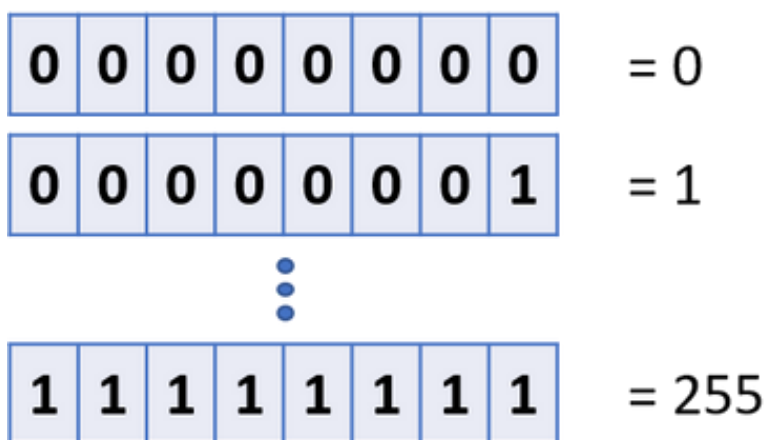
Computer memory and storage devices are often described by how many **bytes** of data they can store.

- A single "**gigabyte**" or "*gig*" is actually 1,073,741,824 bytes! (*that's over 1 billion*)
- Most computers can access **memory addresses** *one byte at a time*.

Here's an example of the *memory layout* of the word "**HELLO**" in a standard character encoding:

Memory Address	Byte Value	ASCII chr
0	72	'H'
1	69	'E'
2	76	'L'
3	76	'L'
4	79	'O'

Below is the [binary](#) layout of bits in a computer, counting from 0, 1, ...up to **255**, which is the largest number that can be stored in a single **byte**.



## Character Encoding

### From numbers to symbols and back (ASCII and friends)

Computer memory is just a sequence of [bytes](#)

- ...and each *byte* is a number from 0-255.

So how does the computer *encode* a bunch of numbers into text strings like "Hello, World"?

- *Simple!* Each *letter* of the alphabet has its own assigned number.
- That goes for *punctuation* and other symbols too. Altogether they're called **characters**.

Here's an example of how you might encode the first three letters of the alphabet:

- 'A' → 65
- 'B' → 66
- 'C' → 67

Those numbers may seem random, but they're the actual values used for A, B, & C in the **ASCII** character set, which is the basic character encoding used by most computers.

- ASCII stands for "American Standard Code for Information Interchange".
- Many other *character encodings* are available today, but ASCII is usually the basis for the English (Latin) alphabet.
- The **Unicode** standard covers *character encoding* for most of the world's written languages.



Python `strings` can be translated *to and from ASCII integers* using the functions:

```
ord(c) # Return the integer Character Encoding for
       # the single-character string 'c'.

chr(n) # Return the string whose Character Encoding
       # is the integer 'n'
```

### Table of Printable ASCII Characters

ord()	chr()
32	SP (Space)
33	! (exclamation mark)
34	" (double quote)
35	# (number sign)
36	\$ (dollar sign)
37	% (percent)
38	& (ampersand)
39	' (single quote)
40	( (left opening parenthesis)
41	) (right closing parenthesis)
42	* (asterisk)
43	+ (plus)
44	, (comma)
45	- (minus or dash)
46	. (dot)
47	/ (forward slash)
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	: (colon)
59	; (semi-colon)
60	< (less than sign)
61	= (equal sign)
62	> (greater than sign)
63	? (question mark)
64	@ (AT symbol)
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[ (left opening bracket)
92	\ (back slash)
93	] (right closing bracket)
94	^ (caret cirumflex)
95	_ (underscore)
96	` (backtick)
97	a
98	b
99	c
100	d
101	e
102	f
103	g

```
104 h
105 i
106 j
107 k
108 l
109 m
110 n
111 o
112 p
113 q
114 r
115 s
116 t
117 u
118 v
119 w
120 x
121 y
122 z
123 { (left opening brace)
124 | (vertical bar)
125 } (right closing brace)
126 ~ (tilde)
```

---

## Computer Simulations

### Creating virtual worlds with code

A computer simulation is code that builds a *model* of something, and lets you play with that model.

In a video game, the *model* could be a **Race Car**, **Football Team**, or a **Fictional Creature**.

Simulations let you explore "virtual" situations, both realistic and imaginary, that might be difficult or impossible to do in the real world.

**Simulations are used in a huge variety of applications:**

- Designing and testing of airplanes and spacecraft
- Video games
- Running flight simulators to train pilots
- Forecasting the weather
- Testing traffic adjustments to design better roadways
- Investigating soil chemistry in agriculture
- Validating electrical circuits
- Testing bridges and buildings before construction



---

## CPU and Peripherals

### Parts of the Computer

The "brain" of the computer that *executes* your code is the Central Processing Unit (CPU) and memory system. When you interact with a computer, it is the **Peripheral** devices that you touch, see, and hear.

**Common Peripherals:**

- LED lights
- Display monitor
- Push buttons
- Keyboard
- Mouse / Trackpad
- Speakers
- Printers (2D, 3D)

Peripherals that bring information **INTO** the CPU are called **Input** devices. Those that send information **OUT** of the computer are **Output** devices. Some peripherals have both *input* and *output* functions.

*Think of the computer peripherals you interact with every day...*

If your brain is your body's CPU, what are its peripherals?

## Debouncing

### Prevent multiple analog triggers

Real-world sensors are almost never perfect

- Analog inputs can easily **bounce** around between one value and the next.

Your program almost always expects a [digital](#) input but the world is mostly [analog](#)!

#### Real-world example

The button press is probably the most common example of bouncing: When a person presses a button they usually expect that the button press will cause an action... **ONE, SINGLE** action.

Unfortunately, the electrical contact may not close instantly or electricity could arc prior to the contact closing.

The single, physical button press could register multiple times in your program.

This can cause *unexpected* behavior - maybe even **dangerous** behavior.

#### The solution:

The best solution is normally just to add a [delay](#).

Waiting in-between readings will give the analog sensor time to settle into an "expected" value.

It can also give a physical contact the opportunity to fully close before reading again!

You can get more complex with your debouncing:

- You could use an average value to smooth out your analog readings.
- You could add a hardware circuit to automate the debounce.

## Debugging

### Fixing your code

#### What is a 'bug'?

When your program doesn't do what you intended, it's called a **bug**.

Actually, most of the time the computer is doing *exactly* what you told it to do! But as a program gets bigger and more complex, it gets harder for us humans to understand. **Debugging** is the process of understanding what the computer is *actually* doing, so you can change the code to do what you *want* it to do.

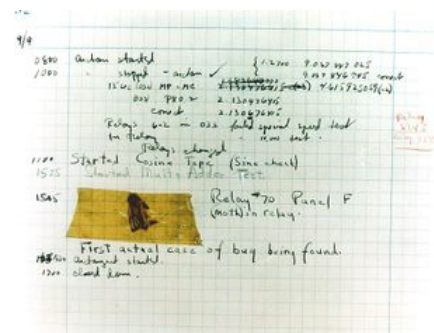
On rare occasions there may be a bug that's a problem in *hardware* rather than your code. One famous bug was noted by renowned computer scientist Admiral Grace Hopper, in her logbook shown here, while doing research using a US Navy computer in 1947.

Yes, that's an *actual* bug - a moth was stuck in one of the computer's mechanical relays!

#### How are bugs fixed?

Apart from checking the computer for moths hanging around, programmers often use additional software, called a *debugger*. Debuggers allow *stepping* through a program and viewing its progress, variables, etc., one line at a time.

Watching how each statement changes the program's [control flow](#) and [variables](#) makes bugs much easier to find and fix.



---

# Divide and Conquer

## Handling complexity

"The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: **divide and conquer**."

-- Bjarne Stroustrup, *Creator of the C++ programming language*

---

# Efficiency

## Writing code that conserves CPU and memory resources.

All computers have limitations in the speed of their [CPUs](#), the amount of *memory* available to store programs and data, etc.

It is very easy to write a program that consumes all the memory in your computer. Just append something to a list a few trillion times; if that's not enough, keep squaring the numbers and you'll soon reach the limits of your machine!

- Along the way you'll find that even on a fast computer there's stuff you can do in your code that negatively impacts the *user experience*.
- Writing *efficient* code is all about **not** doing that!

### How to Write Efficient Code?

The most important step is to carefully consider the *algorithms and data structures* your program needs to get the job done.

- Consider how the number of operations the [CPU](#) must perform might grow as your program runs.
- Consider how the size of [strings](#), [lists](#), [dictionaries](#), and other data structures might grow as well.

*Can it be done with fewer steps? Can it be done with less memory?*

### Don't get carried away

[Readability](#) is almost always more important than *efficiency*. Heed the following Ancient CS Wisdom:

"We should forget about small efficiencies, say about 97% of the time;  
**premature optimization is the root of all evil.**"  
- Sir Tony Hoare (popularized by Donald Knuth)

### But also... don't be a Shlemiel!

Credit to Joel Spolsky for this Yiddish joke which illustrates how algorithms often go wrong.

---

#### Shlemiel the Painter's Algorithm

Shlemiel gets a job as a street painter, painting the dotted lines down the middle of the road. On the first day he takes a can of paint out to the road and finishes 300 yards of the road. "That's pretty good!" says his boss, "you're a fast worker!" and pays him a kopeck.

The next day Shlemiel only gets 150 yards done. "Well, that's not nearly as good as yesterday, but you're still a fast worker. 150 yards is respectable," and pays him a kopeck.

The next day Shlemiel paints 30 yards of the road. "Only 30!" shouts his boss. "That's unacceptable! On the first day you did ten times that much work! What's going on?"

"I can't help it," says Shlemiel. "Every day I get farther and farther away from the paint can!"

---

---

# LCD




## Liquid-Crystal Display (LCD)

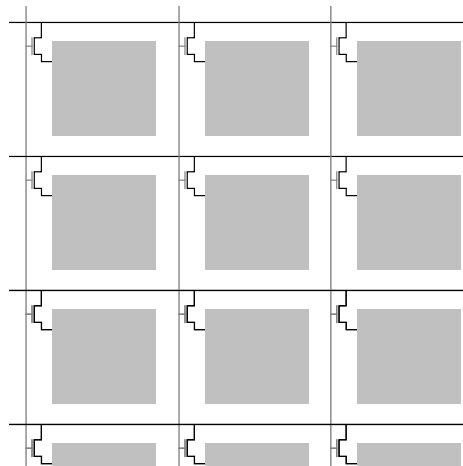
Liquid-Crystal Displays are flat panel hardware peripherals used to output text and graphics.

- You see these in many different electronic products, from TVs to smart watches.

They can display thousands or even *millions* of tiny Red, Green, Blue (RGB) pixels.

The CodeX  `display` is a **TFT-LCD** with 240x240 pixels.

- That's a total of  $240 \times 240 = 57,600$  pixels!
- TFT stands for "Thin Film Transistor"
  - Those *transistors* like switches that turn on/off each of the  RGB color elements in every pixel.
  - You can see them in the upper-left corner of each (gray shaded) pixel in the diagram above.



## LED

### Light Emitting Diode

A **diode** is a fundamental *electronic component*. There are lots of them inside every computer! LEDs are a special kind of diode, packaged in a clear case so the light they emit can shine out.

The **CodeX** has 4 bright color  pixel LEDs plus 6 more red LEDs onboard.

The **CodeBot** has 17 *visible* LEDs (red, green, and yellow) as well as 8 *infrared* LEDs it uses for sensors.



## Modulo and Div Operators

### Sometimes it's all about the remainder

#### Modulo

The % symbol is called **modulo**.

- Sounds like a great Superhero name... or maybe a villain?
- It's nice, really! It gives the *remainder* from a division.

#### Seriously, *Fractions!*?

You may remember learning about the *remainder* when writing improper fractions as mixed numbers. Python can give you the *quotient* and *remainder* separately with // and % operators:

$$\frac{17}{5} = 3R2$$

```
# In Python:
17 // 5 # 3 (quotient)
17 % 5 # 2 (remainder)
```

#### Integer Division

Notice above, when you use the `//` operator you will always get an `int` result. Another name for this is **integer division**. The fractional part is *truncated*.

- It does *not* round up or down... If you need rounding try the `round()` function.

## Monotonic

### Consistently not increasing or not decreasing

**Monotonic is a "fancy word" for a function that is either always not-decreasing or always not-increasing.**

Why does that matter for coding?

To illustrate, consider the `ticks_ms()` function from the `time` module. Note line 6 which is *RISKY*...

```

1 import time
2 start_time = time.ticks_ms()
3 # ...do some time consuming stuff
4 end_time = time.ticks_ms()
5
6 #elapsed = end_time - start_time # RISKY!
7 elapsed = ticks_diff(end_time, start_time)
8
9 print(elapsed)

```

### Why is `ticks_diff()` needed?

The `ticks_ms()` function returns an `integer` that continuously counts up the number of elapsed milliseconds.

- Typically if you save a `ticks_ms()` value, and later save another one, the second will be a larger number.
  - In that case, no problem! Just subtract the first value from the second one and you've got the *diff* right?

### Oops! It's not *monotonic*!

What happens when the value of `ticks_ms()` grows to reach the maximum size of an `integer`?

- It "wraps around" to ZERO!
- Ideally the value would be "monotonic" - meaning it always increases. Then this wouldn't be a problem. Alas, that is not the case!

So, if you risk just doing standard subtraction rather than `ticks_diff()` you might run into the case where you grab the `start_time` value, and then get an `end_time` value that is LESS than the `start_time`!

- The `ticks_diff()` function handles the wrap-around so you always get the proper diff!

## Pins

### Input / Output connections

Pins are physical connection points that allow digital or analog electrical signals to flow from one device to another.

These are places where you can wire to sensors, motors, lights, displays, speakers, and more!

Signals through the pins are generally accessed through software interfaces built into the CPU.

Pins can be used to:

- Output digital signals (LOW or HIGH)
- Read digital signals
- Some can read analog signals (from a sensor)
- Output an analog PWM value

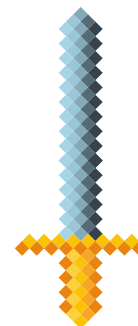
# Pixel

## Elements of a picture

The term **pixel** is short for "picture element". In computer graphics, pixels are the small "dots" used to compose larger images.

When you're looking at your computer or phone screen, you may not be able to **see** the *tiny dots* that make up all the images and text, without a magnifying glass or microscope.

*But they are there!*



# PWM

## Pulse-width Modulation

How can you vary the amount of *electric power* sent to a **motor** or **light**?

- Is it *possible* if you only have a **binary** output?
- All you can do is turn it **fully ON** or **OFF**, right?
- But what if you want to set a light or motor to 50% power?
  - You need **analog** control!

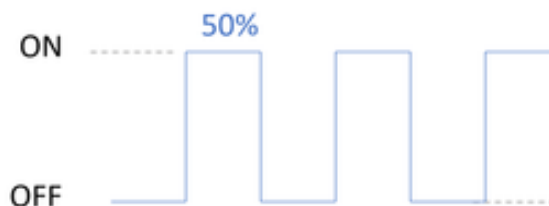
...Actually you **CAN** vary the amount of power using **pulses**!



### Duty Cycle

With Pulse Width Modulation (PWM) you are turning the power ON and OFF rapidly.

- The percentage of **ON** time is called the **Duty Cycle**.
  - Full power = 100%
  - Half power = 50%
  - Zero power = 0%



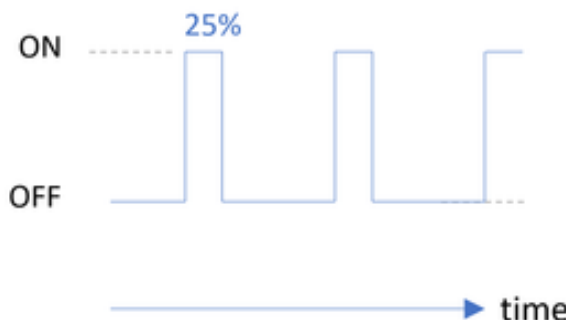
### Frequency

How *rapidly* should you pulse?

The answer depends on the type of device you're controlling.

For example, many PWM applications run at frequencies around **1kHz**.

- That means a pulse is sent every **1ms**
- Pulse-width of 0.1ms → 10% duty cycle.
- Pulse-width of 0.9ms → 90% duty cycle.



**PWM** is used widely to control many types of devices.

The *microcontroller* you're using has built-in hardware to do PWM.

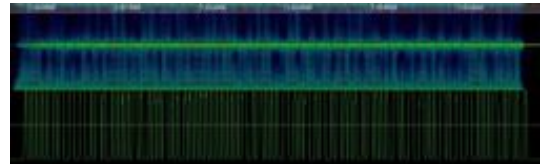
- By adjusting the pulse **frequency** and **duty-cycle** your code has complete control of this powerful capability!

# Radio Frequencies

## Wireless communication for computers

You have probably heard of WiFi and Bluetooth. Those are just two of the **wireless standards** that enable the computers and mobile devices you use every day to communicate.

There are many other standard (and non-standard) wireless methods used in homes, schools, offices, and industrial applications.



Digital radios work by encoding information (like Python strings!) and sending it as *high-frequency* pulses of electrical energy into an antenna, which in turn creates a pulsating ("modulated") *electromagnetic field* that radiates through space and can be received by other antennas.

### Radio Frequency (RF) - what does *high-frequency* mean?

Well, first consider *frequencies* that you can hear - *audio frequencies*.

- For example, a 1000 Hz tone you can play on a speaker. That's kind of a high pitch, but you *might* be able to whistle that high...
- In case you didn't know, **Hz** stands for "Hertz" which is the unit for *frequency* meaning **cycles per second**.

### Standard Bluetooth and WiFi frequency is 2.4 GHz

- That's *giga Hertz* -- 2.4 *billion* pulses per second!
- Okay, that's way too high to whistle :-)

# Readability

## Code that's easy to read and understand

"Programs must be written for **people** to read, and only incidentally for **machines** to execute."

-- H. Abelson and G. Sussman

### Readability is critical to writing good software!

Have you ever written yourself a note and then come back the next day to read it and didn't understand what you wrote?

- That can happen with software too!

Some tips for good readability:

1. Break all repetitive tasks into [functions](#).
2. Give your functions *meaningful* names.
  - `sum_numbers(x, y)` is clear
  - `do_something(x, y)` is not very clear
3. Use names for your [variables](#) that clearly identify their purpose.
  - `frequency = 60` is clear
  - `foo = 60` is not very clear
4. Leave [comments](#) in confusing sections that explain the code.

But it's *my* code and I know what it does!!!

- Always write code as though someone else will need to read it.
  - Most of the time you are not writing software on your own and your co-workers will appreciate *readability*!
  - ...and your *future self* will thank you for it too!

# Reboot

## Start over

When a computer is first powered ON, it's called **Booting Up**<sup>[1]</sup>. Your device has a push-button that *resets* the main CPU, causing it to *start back from the beginning* and run whatever code is loaded again.

1. The term comes from the phrase: "Pull yourself up by your bootstraps" - you know, those loops at the back of boots that help you pull them onto your feet! Think of the computer putting its boots on, after it wakes up first thing in the morning!



---

## Refactoring

### Improving the structure of your code

As you write code there will be occasions when you realize things could be made much better:

- You may notice repeated logic that could be made into a [function](#).
- After writing a sequence of [if conditions](#) you may realize a [dictionary](#) would be more [readable](#).
- You may realize there's a better, simpler, more elegant algorithm to achieve your goals.
- Your code may just become *complex* and messy, in need of restructuring.

**In all of these cases, it is time for *refactoring*.**

What's with the *re*-factoring? Well, it's called that based on the assumption that you've already *factored the problem* to some degree when you wrote the code in the first place. That's all part of [divide and conquer](#), right?

### Tips on Refactoring

1. See the big picture. Take a moment to look over your code at a high-level, and really understand what the "big concepts" are.
2. Get your tests ready.
  - How will you know you haven't broken anything after you refactor?
  - Before you start, come up with a set of tests to prove everything works. Run the tests on the un-refactored program first!
3. Take it one small step at a time, make small changes, and test your code along the way.
  - It really helps if you can keep the code in a "runnable" state as you *evolve* it to a better, *refactored* form.

---

## REPL

### Read Evaluate Print Loop

The **REPL** gives you a way to *interactively* enter commands and view outputs in a text format. You can:

- Call [built-in](#) and user-defined Python [functions](#).
- Check values of [global variables](#).
- See output from [print](#) statements in running code.

A screenshot of a Python REPL interface. The left pane shows the prompt 'Type "help()" for more information.' followed by '>>>' and '>>> c'. Below this is a traceback: 'Traceback (most recent call last): File "<stdin>", line 1, in <module> NameError: name 'c' isn't defined'. The right pane shows 'LOCAL VARIABLES' and 'GLOBAL VARIABLES', both containing 'none'.

```
Type "help()" for more information.
>>>
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' isn't defined
>>>
>>> |
```

**Is "REPL" really a good name for this?**

Well, if you were to code your own "REPL" in Python, it might look something like this:

```
while True:
    # Read a statement from the Keyboard. (press ENTER)
    # Evaluate the statement. (execute the Python code!)
    # Print the result to the Console.
    # Loop (this is a while loop after all...)
```

Besides being a place to see [print](#) statement *output*, the **REPL** is a great way to test out snippets of code, language features, and APIs as you decide how to use them in your code!

## Keyboard Shortcuts

The REPL has some shortcuts to save typing.

- Try using your keyboard [↑](#) and [↓](#) arrow keys to browse previous commands!
- Just hit **ENTER** when you want to execute a command.

## Opening the REPL

In CodeSpace just open the [☰ Console](#) panel at the lower right.

---

# State

## Status of a system with transitions

A **Finite-State Machine**, sometimes called simply a **State Machine**, is a key concept in software and hardware systems.

Your program can only be in **one of a known set** of "states" at any given time. Usually *state* is based on what's in memory - the contents of [variables](#) in your code.

Keeping track of states helps you as a programmer understand and manage your code.

- As code bases grow and more features are added, the "state" of a device can get complicated.
- There may be hundreds of states and multiple different paths to enter and exit each one.
- Each state might have its own set of *conditions* that it is tracking.

Moving between states is called a **transition**.

- The program can **transition** from one state to another when certain [conditions](#) are met.

**An example of a finite-state machine is a traffic light.**

Most traffic lights are three colors:

- Red
- Yellow
- Green

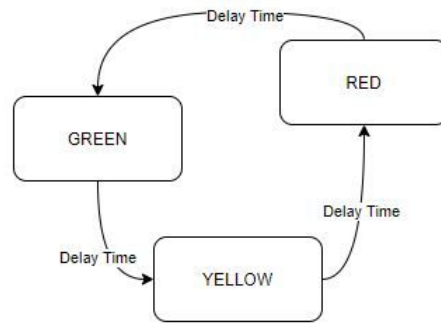
Traffic lights, in the United States of America, generally only have three states:

- GREEN = Traffic can go
- YELLOW = Caution the light will soon be red
- RED = Traffic must stop

The transitions are simple and all transitions occur after a time delay:

- GREEN always transitions to YELLOW
- YELLOW always transitions to RED
- RED always transitions to GREEN

Here is a visual representation of a state machine for a traffic light in the USA:



---

## UI

### User Interface

In computer lingo, the **User Interface** or **UI** is the part of a computer that *humans* interact with directly.

Usual examples would be a **keyboard**, **mouse**, and **screen**.

On the CodeBot and similar *physical computing* devices, the main built-in UI consists of **buttons** and **LEDs**.

---

## USB

### Universal Serial Bus

You may recognize this connection as one that's used for charging many mobile phones, headsets, and other devices. In addition to providing power, USB can also send information!

Connected devices can be *powered* by USB, but also it's the main way that CodeSpace *communicates* with your device's CPU.

---

## CodeSpace

### Advanced Debugging

#### Additional debugger features

CodeSpace includes advanced debugging tools to give more control over a program being debugged.

To enable advanced debugging, use the view menu (View → Show Advanced Debug). Three new buttons will appear on the toolbar:

#### Step Next

- The **Step Next** button will step to the next line, *not* entering function calls.

#### Step Into

- The **Step Into** button will step to the next line, *and* into functions along the way.

#### Step Out

- The **Step Out** button *returns* from the current function, letting you *step* from where it was called.

# Editor Shortcuts

## Keyboard hotkeys to write code faster

Hotkeys in CodeSpace are combinations of keys which complete a task. They are written with a '-' between two key names, like **CTRL-Z**, which you would use by pressing the **Control** key *and* **z** at the same time, then releasing.

### CTRL-X Cut

- Cutting text will **remove** it from the editor, and store it to be Pasted later.
- In the Editor you can **select** text using the mouse, or hold down shift and use the arrow keys on your keyboard.
- **Cut** the selected text using **CTRL-X** on the keyboard, or the *Edit* menu.

### CTRL-C Copy

- Copying text will **not change** what's in the editor, but will store a copy to be Pasted later.
- In the Editor you can **select** text using the mouse, or hold down shift and use the arrow keys on your keyboard.
- **Copy** the selected text using **CTRL-C** on the keyboard, or the *Edit* menu.

### CTRL-V Paste

- Put the editor cursor where you want to insert text, and press **CTRL-V** or use the *Edit* menu to **Paste**.

### CTRL-Z Undo

- **Undo** the last change to your code using **CTRL-Z** on the keyboard, or the **Undo** toolbar button.

### CTRL-F Search

- **Search** for text in your program using **CTRL-F** on the keyboard, or the **Search** toolbar button.

### CTRL-H Replace

- **Search and Replace** text in your program using **CTRL-H** on the keyboard.

### TAB or SHIFT-TAB Indent

- **Indent** or **Unindent** a selected block of text in your program.

### CTRL-/ Comment Toggle

- **Comment-out** or **un-Comment** the current line or selected block of code using **CTRL-/**

---

# File System

## How to Use the CodeSpace File System

The CodeSpace filesystem can be interacted with in two different ways:

- Through the [UI](#), which will be detailed here, or
- Programmatically through [File Operations](#).

## File Browser

Many file actions occur in the file browser menu.

- The file browser gives you a visual representation of the files in your filesystem and allows you to perform operations on them.

The file browser can be accessed with two simple steps:

1. Click the 'File' button in the top left hand corner of your window.
2. Click 'Browse Files' in the dropdown which was opened in the previous step.



## Open

In the 'Browse Files' menu, to open a file either:

- Double click the file, *or*
- Select the file with a single click, then click 'Open' in the bottom right hand corner of the 'Browse Files' window.

## File Information Window

Within the file browser, each file has its own 'File Information Window'.

- From the 'File Information Window', you can delete or rename a file.

In the 'Browse Files' menu, to open the 'File Information Window':

1. Click the pencil icon on the right side of filename.

## Delete

In the 'File Information Window', to delete a file:


1. Click the 'Delete' button in the top right corner of the window.

## Rename

In the 'File Information Window', to rename a file:

1. Type your desired filename in the 'Rename' text box.
2. Click 'OK'

## Tabs

When you open a file using the , a new tab is created.

Tabs provide an interface for switching between and closing your opened files.

---

# Syntax Highlighting

## Why the code has colors

Wondering how to add colors to the Text Editor panel?

No problem!

The Editor **automatically** colors Python keywords and other *language syntax* to make your code easier to read. This is called *syntax highlighting*, and it's a common feature of text editors built for coding.

**Syntax?** That's just a fancy word for the "rules" about how words and phrases go together to form a language. Computer languages and human languages alike have *syntax* rules to follow if you want to be understood!

---

# LiftOff Peripherals

## Servos

### DC Servo Motors

#### What is a servo?

A servo is more than just a motor. It contains:

- A DC motor

- A controller circuit
- An internal feedback mechanism
- An amplifier (or gearbox)

### How do I make the servo go?

Servo motors require an **analog control signal** to operate.

- You can send an analog control signal using [PWM!](#)

Nearly all servos operate with a **50 Hertz (Hz)** control signal.

- 50 Hz = 50 times per second = 20 millisecond analog period.

50 Hz became a standard long ago due to the simplicity of the hardware design.

### Types of Servos

- The **360 Continuous Rotation Servo** which can rotate continuously backward and forward.

Duty Cycle (ms)	Speed	Direction
0.5	100%	Clockwise
0.75	75%	Clockwise
1.0	50%	Clockwise
1.25	25%	Clockwise
1.5	0%	Stopped
1.75	25%	Counterclockwise
2.0	50%	Counterclockwise
2.25	75%	Counterclockwise
2.5	100%	Counterclockwise

- The **180 Positional Servo** which can move to a specified position and hold in place.

Duty Cycle (ms)	Position
0.5	90 Degrees Clockwise
1.0	45 Degrees Clockwise
1.5	Centered
2.0	45 Degrees Counterclockwise
2.5	90 Degrees Counterclockwise

### Positional Servos Stay in Position

There is no OFF position for the 180 servo like there is for a 360 servo.

- The 180 servo is always working to stay in position
- If you push it either direction it will always come back to its set position