Basic Motion

Welcome to JetBot's browser based programming interface! This document is called a *Jupyter Notebook*, which combines text, code, and graphic display all in one! Pretty neat, huh? If you're unfamiliar with *Jupyter* we suggest clicking the `Help` drop down menu in the top toolbar. This has useful references for programming with *Jupyter*.

In this notebook, we'll cover the basics of controlling JetBot.

Importing the Robot class

To get started programming JetBot, we'll need to import the `Robot` class. This class allows us to easily control the robot's motors! This is contained in the `jetbot` package.

> If you're new to Python, a *package* is essentially a folder containing code files. These code files are called *modules*.

To import the `Robot` class, highlight the cell below and press `ctrl + enter` or the `play` icon above. This will execute the code contained in the cell

In [1]:
```
from jetbot import Robot
```

Now that we've imported the `Robot` class we can initialize the class *instance* as follows.

In [2]:
```
robot = Robot()
```

Commanding the robot

Now that we've created our `Robot` instance we named "robot", we can use this instance to control the robot. To make the robot spin counterclockwise at 30% of it's max speed we can call the following

> WARNING: This next command will make the robot move! Please make sure the robot has clearance.

In [3]:
```
robot.left(speed=0.3)
```

Cool, you should see the robot spin counterclockwise!

> If your robot didn't turn left, that means one of the motors is wired backwards! Try powering down your robot and swapping the terminals that the `red` and `black` cables of the incorrect motor.
>
> REMINDER: Always be careful to check your wiring, and don't change the wiring on a running system!

Now, to stop the robot you can call the `stop` method.

In [4]:
```
robot.stop()
```

Maybe we only want to run the robot for a set period of time. For that, we can use the Python `time` package.

In [5]:
```
import time
```

This package defines the `sleep` function, which causes the code execution to block for the specified number of seconds before running the next command. Try the following to make the robot turn left only for half a second.

In [6]:
```
robot.left(0.3)
time.sleep(0.5)
robot.stop()
```

Great. You should see the robot turn left for a bit and then stop.

> Wondering what happened to the `speed=` inside the `left` method? Python allows us to set function parameters by either their name, or the order that they are defined (without specifying the name).

The `BasicJetbot` class also has the methods `right`, `forward`, and `backward`. Try creating your own cell to make the robot move forward at 50% speed for one second.

Create a new cell by highlighting an existing cell and pressing `b` or the `+` icon above. Once you've done that, type in the code that you think will make the robot move forward at 50% speed for one second.

Controlling motors individually

Above we saw how we can control the robot using commands like `left`, `right`, etc. But what if we want to set each motor speed individually? Well, there are two ways you can do this

The first way is to call the `set_motors` method. For example, to turn along a left arch for a second we could set the left motor to 30% and the right motor to 60% like follows.

In [7]:
```
robot.set_motors(0.3, 0.6)
time.sleep(1.0)
robot.stop()
```

Great! You should see the robot move along a left arch. But actually, there's another way that we could accomplish the same thing.

The `Robot` class has two attributes named `left_motor` and `right_motor` that represent each motor individually. These attributes are `Motor` class instances, each which contains a `value` attribute. This `value` attribute is a traitlet (https://github.com/ipython/traitlets) which generates `events` when assigned a new value. In the motor class, we attach a function that updates the motor commands whenever the value changes.

So, to accomplish the exact same thing we did above, we could execute the following.

In [8]:
```
robot.left_motor.value = 0.3
robot.right_motor.value = 0.6
time.sleep(1.0)
robot.left_motor.value = 0.0
robot.right_motor.value = 0.0
```

You should see the robot move in the same exact way!

Link motors to traitlets

A really cool feature about these traitlets (https://github.com/ipython/traitlets) is that we can also link them to other traitlets! This is super handy because Jupyter Notebooks allow us to make graphical `widgets` that use traitlets under the hood. This means we can attach our motors to `widgets` to control them from the browser, or just visualize the value.

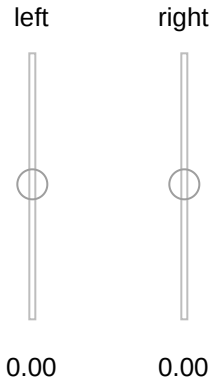
To show how to do this, let's create and display two sliders that we'll use to control our motors.

In [9]:
```
import ipywidgets.widgets as widgets
from IPython.display import display

# create two sliders with range [-1.0, 1.0]
left_slider = widgets.FloatSlider(description='left', min=-1.0, max=1.0,
right_slider = widgets.FloatSlider(description='right', min=-1.0, max=1.

# create a horizontal box container to place the sliders next to each ot
slider_container = widgets.HBox([left_slider, right_slider])

# display the container in this cell's output
display(slider_container)
```



You should see two `vertical` sliders displayed above.

> HELPFUL TIP: In Jupyter Lab, you can actually "pop" the output of cells into entirely separate window! It will still be connected to the notebook, but displayed separately. This is helpful if we want to pin the output of code we executed elsewhere. To do this, right click the output of the cell and select `Create New View for Output`. You can then drag the new window to a location you find pleasing.

Try clicking and dragging the sliders up and down. Notice nothing happens when we move the sliders currently. That's because we haven't connected them to motors yet! We'll do that by using the `link` function from the traitlets package.

In [10]:
```
import traitlets

left_link = traitlets.link((left_slider, 'value'), (robot.left_motor, 'v
right_link = traitlets.link((right_slider, 'value'), (robot.right_motor,
```

Now try dragging the sliders (slowly at first). You should see the respective motor turn!

The `link` function that we created above actually creates a bi-directional link! That means, if we set the motor values elsewhere, the sliders will update! Try executing the code block below

In [11]:
```
robot.forward(0.3)
time.sleep(1.0)
robot.stop()
```

You should see the sliders respond to the motor commands! If we want to remove this connection we can call the `unlink` method of each link.

In [12]:
```
left_link.unlink()
right_link.unlink()
```

But what if we don't want a *bi-directional* link, let's say we only want to use the sliders to display the motor values, but not control them. For that we can use the `dlink` function. The left input is the `source` and the right input is the `target`

In [13]:
```
left_link = traitlets.dlink((robot.left_motor, 'value'), (left_slider, '
right_link = traitlets.dlink((robot.right_motor, 'value'), (right_slider
```

Now try moving the sliders. You should see that the robot doesn't respond. But when set the motors using a different method, the sliders will update and display the value!

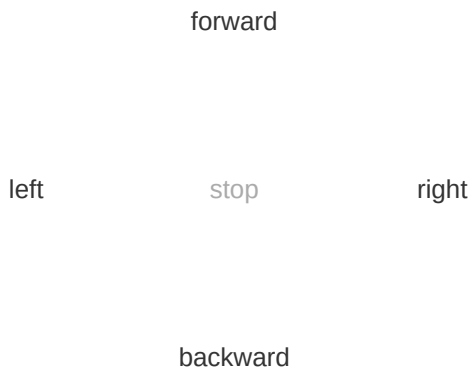
Attach functions to events

Another way to use traitlets, is by attaching functions (like `forward`) to events. These functions will get called whenever a change to the object occurs, and will be passed some information about that change like the `old` value and the `new` value.

Let's create and display some buttons that we'll use to control the robot.

In [14]:
```
# create buttons
button_layout = widgets.Layout(width='100px', height='80px', align_self=
stop_button = widgets.Button(description='stop', button_style='danger',
forward_button = widgets.Button(description='forward', layout=button_lay
backward_button = widgets.Button(description='backward', layout=button_l
left_button = widgets.Button(description='left', layout=button_layout)
right_button = widgets.Button(description='right', layout=button_layout)

# display buttons
middle_box = widgets.HBox([left_button, stop_button, right_button], layo
controls_box = widgets.VBox([forward_button, middle_box, backward_buttor
display(controls_box)
```



You should see a set of robot controls displayed above! But right now they wont do anything. To do that we'll need to create some functions that we'll attach to the button's `on_click` event.

In [15]:
```
def stop(change):
    robot.stop()

def step_forward(change):
    robot.forward(0.4)
    time.sleep(0.5)
    robot.stop()

def step_backward(change):
    robot.backward(0.4)
    time.sleep(0.5)
    robot.stop()

def step_left(change):
    robot.left(0.3)
    time.sleep(0.5)
    robot.stop()

def step_right(change):
    robot.right(0.3)
    time.sleep(0.5)
    robot.stop()
```

Now that we've defined the functions, let's attach them to the on-click events of each button

In [16]:
```
# link buttons to actions
stop_button.on_click(stop)
forward_button.on_click(step_forward)
backward_button.on_click(step_backward)
left_button.on_click(step_left)
right_button.on_click(step_right)
```

Now when you click each button, you should see the robot move!

Heartbeat Killswitch

Here we show how to connect a 'heartbeat' to stop the robot from moving. This is a simple way to detect if the robot connection is alive. You can lower the slider below to reduce the period (in seconds) of the heartbeat. If a round-trip communication between browser cannot be made within two heartbeats, the ' `status` ' attribute of the heartbeat will be set `dead` . As soon as the connection is restored, the `status` attribute will return to `alive` .

In [17]:
```
from jetbot import Heartbeat

heartbeat = Heartbeat()

# this function will be called when heartbeat 'alive' status changes
def handle_heartbeat_status(change):
    if change['new'] == Heartbeat.Status.dead:
        robot.stop()

heartbeat.observe(handle_heartbeat_status, names='status')

period_slider = widgets.FloatSlider(description='period', min=0.001, ma
traitlets.dlink((period_slider, 'value'), (heartbeat, 'period'))

display(period_slider, heartbeat.pulseout)
```

period 0.50

1640323693.6349401

Try executing the code below to start the motors, and then lower the slider to see what happens. You can also try disconnecting your robot or PC.

In [18]:
```
robot.left(0.2)

# now lower the `period` slider above until the network heartbeat can't
```

Conclusion

That's it for this example notebook! Hopefully you feel confident that you can program your robot to move around now :)