Lesson 3 Project: Backyard Flyer

1. Lesson Introduction

<https://www.youtube.com/watch?time_continue=1&v=z48qy6lGU5M>

1. Lesson Overview

<https://www.youtube.com/watch?v=wIRJPPH2gkc>

**Lesson Overview**

In this lesson you'll learn how to control a drone with code. More specifically, you'll learn how to write *event-driven* code that controls a drone.

This event-driven programming paradigm will allow you to write drone programs that respond dynamically to a drone's changing and unpredictable world.

**Outline**

1. **Simulator Exploration** - first, you will become familiar with the drone simulator you'll be using throughout this program.
2. **Flight Computer Programming** - you'll review the roles of a drone's flight computer and autopilot so you understand how the flight computer code you'll be writing actually communicates with the drone's low-level autopilot.
3. **Event Driven Programming** - you'll explore the concepts underlying event driven programming by reading and then modifying Python code for an EventDrivenChatBot class.
4. **A Simple Flight Plan** - you'll see what event-driven programming looks like in the context of a real (but simple) flight plan.
5. **Phases of Flight** - you'll learn how we represent the "lifecycle" of a flight plan with a finite state machine.
6. **Backyard Flyer** - At the end of this lesson you'll actually implement your own flight plan within the event-driven programming paradigm you've been learning about.
7. Simulator Demonstration

<https://www.youtube.com/watch?time_continue=26&v=8zpkGojsw6g>

1. Simulator Exploration—Manual Flight

**Simulator Exploration - Manual**

Before you learn about new programming paradigms, you might want to get setup with the Udacity drone simulator.

**Download and Installation**

You can [**download the simulator here**](https://github.com/udacity/FCND-Simulator-Releases/releases).

Once you've downloaded the latest release, unzip the file and open the simulator by double-clicking on the app/executable.

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/b5a62b08-513b-4e1c-8041-b1d37c105f3a)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/b5a62b08-513b-4e1c-8041-b1d37c105f3a)**

**[The Udacity drone simulator](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/b5a62b08-513b-4e1c-8041-b1d37c105f3a)**

**Arming**

Arming implies that the power is now being supplied to the motors and the drone is ready to fly. Think of it like using your keys to start your car before you hit the gas! You always need to arm your drone first before you begin to fly. Once you click on arming, you'll notice that the propellers start to spin. And once you click Disarm, the motors stop. **Remember to always Arm your drone before you can fly and Disarm the drone when not flying it.**

**QUESTION 1 OF 3**

Were you able to arm your Drone?

* Yes! The propellers are spinning and it's ready to take off.
* 

No, I can't get it working.

SUBMIT

**Manual vs Guided**

As the name suggests, when you are flying the drone in Manual mode, you are the pilot. If it's a real drone, then you'll be using an RC controller. In the simulator, you can use your keyboard to fly the drone. Note that even in Manual mode, the drone is doing a lot of the work "autonomously" as it converts your high-level commands into rotor rotation rates.

Guided mode is used when you want the drone to perform tasks autonomously without any human control. You can do this either by sending commands to the drone via console or by writing a script to execute those commands depending on the mission requirements.

For this project, you'll be writing a program that will lead the drone to execute a mission. In order for the drone to run the mission on its own, it needs to be in Guided mode.

**QUESTION 2 OF 3**

In the simulator, you can apply upward and downward thrust using the Space and C keys on your keyboard.

What happens when you try to use these keys when the drone is in Guided mode?

* 

The drone goes up and down.

* 

The commands have no effect.

SUBMIT

You can manually control the drone using the following key commands:

| **Key(s)** | **Control** |
| --- | --- |
| W/S | move forward / back |
| A/D | move left / right |
| Q/E | turn left/ right |

Try using these controls to take off, fly in a square shape, and then land.

**QUESTION 3 OF 3**

Were you able to take off, fly in a square, and then land?

* 

Yes

* 

No

SUBMIT

NEXT

1. Flight Computer Programming

<https://www.youtube.com/watch?v=8TFLgWGyfrg>

**Connection to Code**

When writing your solution to this project, you will write flight computer code by filling out methods in a class called BackyardFlyer (which inherits from a Udacity-provided Drone class). The code you write will have several responsibilities, but two of the most important will involve:

1. **Setting the drone's target position (for the autopilot to follow)** - You'll do this by using the cmd\_position method from the Udacity-provided Droneclass.
2. **Responding to messages from the autopilot** - Your BackyardFlyer class will be sent messages from the autopilot to let it know when something about the drone has changed. For example, the autopilot will send a LOCAL\_POSITIONmessage whenever the drone's local position changes. You'll be responsible for *registering* certain functions (called "callback functions") to respond to these messages. For example, if you want to call some function named local\_position\_callback whenever you receive a LOCAL\_POSITION update, you would write code that says
3. self.register\_callback(MsgID.LOCAL\_POSITION, self.local\_position\_callback)

This code will make more sense by the end of this lesson.

**A Note on Prerequisites**

**Python and Object Oriented Programming**

In this Nanodegree, we assume that you know Python. In particular, we assume you are comfortable with object oriented programming (OOP) in Python.

If you need a refresher on Python or OOP, consider reviewing Udacity's [**Programming Foundations with Python**](https://www.udacity.com/course/programming-foundations-with-python--ud036)

**Git and Github**

We will be using git and Github throughout this Nanodegree and we assume you are familiar with the basics of of these technologies. If you find yourself confused by git or Github, take a look at Udacity's [**Version Control with Git**](https://www.udacity.com/course/version-control-with-git--ud123).

1. Environment Setup

**Environment Setup (~30 minutes)**

Before you can program flight plans for the simulator, there's a bit of setup that's required first.

You are welcome to skip this section if you are taking the free preview and do not want to command the drone with Python code or complete the Backyard Flyer project.

**Setup Instructions (abbreviated)**

Read through the instructions below. If these commands look familiar to you, then you should use these VERY abbreviated instructions to get yourself set up.

1. [**download miniconda**](https://conda.io/miniconda.html) and then install by opening the file/app that you download.
2. git clone https://github.com/udacity/FCND-Term1-Starter-Kit.git to clone the starter kit and then cd FCND-Term1-Starter-Kit into that directory. If you have a windows machine, you must rename meta\_windows\_patch.yml to meta.yml as well.
3. conda env create -f environment.yml to create the miniconda environment: *this took me 20 minutes to run due to the large number of installs required.*
4. source activate fcnd to activate the environment (you'll need to do this whenever you want to work in this environment).

**Setup Instructions (detailed)**

**0. Install git (if you don't already have it)**

You likely already have git available as a command line application, but if you don't you should follow the **[git setup instructions](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git" \t "_blank)** that are appropriate for your machine.

**1. Download and Install Miniconda**

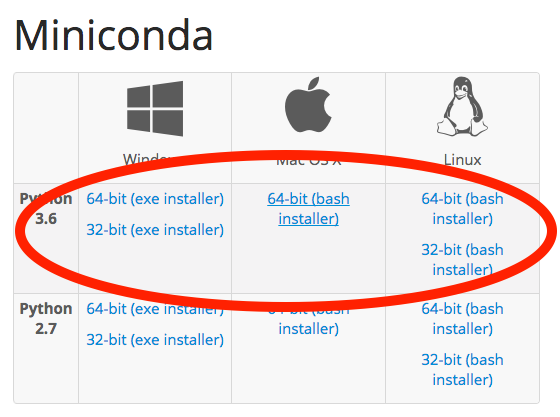
We will be using Miniconda to install Python. This will help ensure that the version of Python (and all the required libraries) that you work with are exactly what they're supposed to be. This should help avoid dependency problems.

1. Go to the **[miniconda distribution page](https://conda.io/miniconda.html" \t "_blank)** and download the version of Miniconda that is right for your computer.
2. Once the download is complete, double-click the app/file to install. If this doesn't work, you can use the terminal to execute the miniconda shell script. cd into the directory with the downloaded script and then run the following:
3. bash <FILENAME>

for me (on a mac), that was:

bash Miniconda3-latest-MacOSX-x86\_64.sh

If you are prompted with a question like "Do you wish the installer to prepend the Miniconda3 install location to PATH..." you should say yes.

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/6d8be91e-0df2-4558-8723-727cf9b64243)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/6d8be91e-0df2-4558-8723-727cf9b64243)**

**[Make sure you download Miniconda for Python 3.6](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/6d8be91e-0df2-4558-8723-727cf9b64243)**

**2. Get the Flying Car "Starter Kit"**

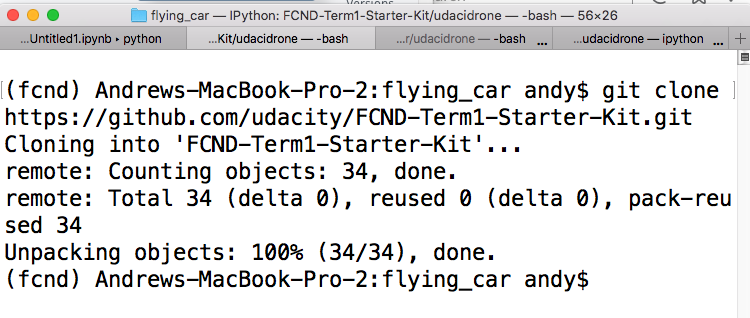
We're providing you with a "starter kit" that contains the required dependencies you'll need to get up and running.

Open a terminal window and run the following commands:

git clone https://github.com/udacity/FCND-Term1-Starter-Kit.git

cd FCND-Term1-Starter-Kit

**If you have a windows machine** you must rename meta\_windows\_patch.yml to meta.yml as well.

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/6d8be91e-0df2-4558-8723-727cf9b64243)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/6d8be91e-0df2-4558-8723-727cf9b64243)**

**3. Create the Environment (~20-30 mins of download required)**

From the FCND-Term1-Starter-Kit, run the following command:

conda env create -f environment.yml

Running this command will create a new conda environment that is provisioned with all libraries you need to be successful in this program.

**This step downloads and installs all the required libraries. It may take a while!**It took 20 mins on my machine (with a fast internet connection).

NOTE: If the above command fails due to internet issues or timed out HTTP request then remove the partially built environment using the following command (then run the above create command again):

conda env remove -n fcnd

Verify that the FCND (listed fcnd) environment was created in your environments:

conda info --envs

Cleanup downloaded libraries (remove tarballs, zip files, etc):

conda clean -tp

**4. Activate the Conda Environment**

Now that you have created an environment, in order to use it, you will need to activate it.

**This must be done each time you begin a new working session i.e. open a new terminal window.**

For OS X and Linux:

$ source activate fcnd

For Windows depending on shell either:

$ source activate fcnd

or

$ activate fcnd

That's it. Now all of the fcnd libraries are available to you.

1. Simulator Exploration-Programmatic Flight

**Simulator Exploration - Programmatic Flight**

Now it's time to actually control the drone with your own Python code.

NOTE - This section assumes you've completed the steps to setup your environment on the previous page.

Are you ready to start controlling the drone with code? Make sure you've completed the tasks below before you begin.

Task List

* 

I've downloaded the drone simulator and commanded the drone with my keyboard.

* 

I've downloaded and installed Miniconda.

* 

I've cloned the FCND Starter Kit repository.

* 

I've created the fcnd conda environment in the starter kit directory.

**Communicating with the Drone using the API**

In order to use the Drone API to communicate with the simulator. You first need to open the simulator.

Next, you will need to activate the environment via your terminal using the following command

source activate fcnd

Now the drone can be manually started from a python / ipython shell! From the terminal start ipython:

ipython

Now you can initialize the drone with the following commands:

**from** udacidrone **import** Drone

**from** udacidrone.connection **import** MavlinkConnection

conn = MavlinkConnection('tcp:127.0.0.1:5760', threaded=**True**)

drone = Drone(conn)

drone.start()

Now take control of the drone and arm the rotors. Briefly pause in between executing code snippets as running them in immediate succession can cause strange drone behavior within the simulator.

drone.take\_control()

drone.arm()

Now set the drone's "home position"

drone.set\_home\_position(drone.global\_position[0],

drone.global\_position[1],

drone.global\_position[2])

And now you can take off (to a height of 3 meters)!

drone.takeoff(3)

Once you're in the air, you can fly around by commanding the drone to waypoints.

drone.cmd\_position(5,0,3,0)

**Drone Commands**

There are many commands you can issue to the drone through this API. Some of them include...

start(): Start receiving messages from the drone. If the connection is not threaded, this will block the code

stop(): Terminate the connection with the drone and close the telemetry log

take\_control(): Set the command mode of the quad to guided.

release\_control(): Set the command mode of the quad to manual.

arm(): Arms the motors of the quad, the rotors begin spinning. The drone cannot takeoff until armed.

disarm(): Disarms the motors of the quad. The quadcopter cannot be disarmed in the air.

cmd\_position(north, east, down, heading): Command the vehicle to travel to the local position (north, east, down). Also commands the quad to maintain a specified heading.

**QUIZ QUESTION**

Were you able to connect to the simulator and control it with Python code?

* Yes
* 

No

SUBMIT

**Full API Documentation**

To learn more about this Drone class and the provided API, check out the [**README in the Github repo**](https://github.com/udacity/udacidrone)

1. The problems with Sequential Execution

<https://www.youtube.com/watch?v=TCz0PkLoxgU>

**So what is Event Driven Programming?**

We'll be talking about this more in the sections that follow, but I'd like to give a quick summary here:

*Event-driven programming is a programming paradigm in which the flow of execution is determined by external****events****rather than a pre-defined sequence of steps.*

When would you use EDP?

A drone is one example. But EDP is also the dominant paradigm used in: graphical user interfaces, where programmers specify how the application should respond to user actions (events).

You might also use event-driven programming when writing a chatbot. And the example you'll see in the next section uses an EventDrivenChatBot class to demonstrate some of the key concepts in EDP.

1. Event Driven Programming

**Event Driven Programming - Chatbot**

A chatbot is a perfect way to demonstrate event driven programming since we can't dictate in advance the "correct" sequence of statements a chatbot should make.

Instead, a chatbot should respond to "events" which come in the form of messages from the chatbot's conversation partner.

**Goals for this activity**

1. (re)familiarize yourself with how Python handles classes and object oriented programming.
2. gain familiarity with what types of problems event driven programming is good for.
3. write simple code that works within this event driven framework

**TODO 1 - Read through and understand the EventDrivenChatBot class**

Read through the code below until you have a prediction for what will happen when you **execute** this code by pressing the Test Run button.

NOTE - the last few lines of the code below is where the EventDrivenChatBotclass is instantiated and its methods are called.

from datetime import datetime

import time

class EventDrivenChatBot:

def \_\_init\_\_(self):

# accepted\_messages maps incoming messages to

# list of callback functions

self.accepted\_messages = {}

# time of instantiation

self.birth\_time = datetime.now()

# "registering" all callbacks

self.register\_callback("hi",

self.respond\_to\_greeting)

self.register\_callback("hi2",

self.respond\_to\_greeting)

self.register\_callback("bye",

self.respond\_to\_departure)

self.register\_callback("age?",

self.respond\_to\_age\_request)

self.register\_callback("age?",

self.respond\_to\_age\_request\_detailed)

def register\_callback(self, message, callback):

"""

Registers a callback to a message.

"""

if message not in self.accepted\_messages:

self.accepted\_messages[message] = []

self.accepted\_messages[message].append(callback)

def respond\_to\_greeting(self):

print("Hello!")

def respond\_to\_departure(self):

print("Nice chatting with you!")

def respond\_to\_age\_request(self):

age = datetime.now() - self.birth\_time

print("I am", age.seconds, "seconds old.")

def respond\_to\_age\_request\_detailed(self):

age = datetime.now() - self.birth\_time

micros = age.microseconds

print("Technically, I'm", age.seconds, "seconds and",

micros, "microseconds old")

def handle\_message(self, message):

if message not in self.accepted\_messages:

print("sorry, I don't understand", message)

else:

callbacks = self.accepted\_messages[message]

for callback in callbacks:

callback()

bot = EventDrivenChatBot()

bot.handle\_message("hi")

time.sleep(4.8)

bot.handle\_message("age?")

bot.handle\_message("bye")

**TODO 2 - Run the Code**

Press the Test Run button above and look at the output. How does it compare to what you expected to see? What happens between calling the handle\_messagemethod and actually printing the chatbot's response?

**TODO 3 - Modify the Code**

Once you feel like you understand how this code works, modify EventDrivenChatBotso that the FIRST time it receives the message "age?" it calls respond\_to\_age\_request but all subsequent "age?" messages should trigger a call to respond\_to\_age\_request\_detailed.

You can find my solution in the next section of this lesson.

1. Event Driven Programming Explained

**Event Driven Programming Explained**

The code below is my solution to the last TODO in the previous section. Below that you'll find a video walkthrough of the EventDrivenChatBot code.

**Solution Overview**

There are many ways to solve this problem, but most solutions probably involve modeling the "state" of the chatbot as either

1. has NOT yet seen the message "age?" OR
2. HAS seen the message "age?"

I implemented this by making three modifications...

1. In \_\_init\_\_ I added a state variable has\_been\_asked\_age which is initially set to False
2. I define a function called handle\_age\_request which in turn calls either respond\_to\_age\_request or respond\_to\_age\_request\_detailed depending on the STATE of the bot.
3. Register a callback that associated handle\_age\_request to the "ask?"message. Note that I also removed two calls to register\_callback that were there previously.

from datetime import datetime

import time

class EventDrivenChatBot:

def \_\_init\_\_(self):

self.accepted\_messages = {}

# 1. ADDED THIS "STATE" VARIABLE

self.has\_been\_asked\_age = False

self.birth\_time = datetime.now()

# "registering" all callbacks

self.register\_callback("hi",

self.respond\_to\_greeting)

self.register\_callback("bye",

self.respond\_to\_departure)

# 3. USING handle\_age\_request TO DISPATCH

# RESPONSES TO "age?"

self.register\_callback("age?",

self.handle\_age\_request)

def register\_callback(self, message, callback):

"""

Registers a callback to a message.

"""

if message not in self.accepted\_messages:

self.accepted\_messages[message] = []

self.accepted\_messages[message].append(callback)

def respond\_to\_greeting(self):

print("Hello!")

def respond\_to\_departure(self):

print("Nice chatting with you!")

# 2. ADD DISPATCH POINT FOR PROCESSING "age?" MESSAGE

def handle\_age\_request(self):

if not self.has\_been\_asked\_age:

self.has\_been\_asked\_age = True

self.respond\_to\_age\_request()

else:

self.respond\_to\_age\_request\_detailed()

def respond\_to\_age\_request(self):

age = datetime.now() - self.birth\_time

print("I am", age.seconds, "seconds old.")

def respond\_to\_age\_request\_detailed(self):

age = datetime.now() - self.birth\_time

micros = age.microseconds

print("Technically, I'm", age.seconds, "seconds and",

micros, "microseconds old")

def handle\_message(self, message):

if message not in self.accepted\_messages:

print("sorry, I don't understand", message)

else:

callbacks = self.accepted\_messages[message]

for callback in callbacks:

callback()

bot = EventDrivenChatBot()

bot.handle\_message("hi")

time.sleep(1.3)

bot.handle\_message("age?")

print("---No chatbot, let me ask you that again...")

bot.handle\_message("age?")

<https://www.youtube.com/watch?v=f_GPA9ULqJU>

1. A Simple Flight Plan

<https://www.youtube.com/watch?v=Lq4DmvlsJAI>

**up\_and\_down.py**

I've included all the code from up\_and\_down.py below for your reference.

**import** time

**from** enum **import** Enum

**import** numpy **as** np

**from** udacidrone **import** Drone

**from** udacidrone.connection **import** MavlinkConnection

**from** udacidrone.messaging **import** MsgID

**class** **Phases**(Enum):

MANUAL = 0

ARMING = 1

TAKEOFF = 2

LANDING = 3

DISARMING = 4

**class** **UpAndDownFlyer**(Drone):

**def** **\_\_init\_\_**(self, connection):

super().\_\_init\_\_(connection)

self.target\_position = np.array([0.0, 0.0, 0.0])

self.in\_mission = **True**

*# initial state*

self.flight\_phase = Phases.MANUAL

*# register all your callbacks here*

self.register\_callback(MsgID.LOCAL\_POSITION,

self.local\_position\_callback)

self.register\_callback(MsgID.LOCAL\_VELOCITY,

self.velocity\_callback)

self.register\_callback(MsgID.STATE,

self.state\_callback)

**def** **local\_position\_callback**(self):

**if** self.flight\_phase == Phases.TAKEOFF:

*# coordinate conversion*

altitude = -1.0 \* self.local\_position[2]

*# check if altitude is within 95% of target*

**if** altitude > 0.95 \* self.target\_position[2]:

self.landing\_transition()

**def** **velocity\_callback**(self):

**if** self.flight\_phase == Phases.LANDING:

**if** ((self.global\_position[2] - self.global\_home[2] < 0.1) **and**

abs(self.local\_position[2]) < 0.01):

self.disarming\_transition()

**def** **state\_callback**(self):

**if** **not** self.in\_mission:

**return**

**if** self.flight\_phase == Phases.MANUAL:

self.arming\_transition()

**elif** self.flight\_phase == Phases.ARMING:

**if** self.armed:

self.takeoff\_transition()

**elif** self.flight\_phase == Phases.DISARMING:

**if** **not** self.armed:

self.manual\_transition()

**def** **arming\_transition**(self):

print("arming transition")

self.take\_control()

self.arm()

*# set the current location to be the home position*

self.set\_home\_position(self.global\_position[0],

self.global\_position[1],

self.global\_position[2])

self.flight\_phase = Phases.ARMING

**def** **takeoff\_transition**(self):

print("takeoff transition")

target\_altitude = 3.0

self.target\_position[2] = target\_altitude

self.takeoff(target\_altitude)

self.flight\_phase = Phases.TAKEOFF

**def** **landing\_transition**(self):

print("landing transition")

self.land()

self.flight\_phase = Phases.LANDING

**def** **disarming\_transition**(self):

print("disarm transition")

self.disarm()

self.flight\_phase = Phases.DISARMING

**def** **manual\_transition**(self):

print("manual transition")

self.release\_control()

self.stop()

self.in\_mission = **False**

self.flight\_phase = Phases.MANUAL

**def** **start**(self):

self.start\_log("Logs", "NavLog.txt")

print("starting connection")

super().start()

self.stop\_log()

**if** \_\_name\_\_ == "\_\_main\_\_":

conn = MavlinkConnection('tcp:127.0.0.1:5760',

threaded=**False**,

PX4=**False**)

drone = UpAndDownFlyer(conn)

time.sleep(2)

drone.start()

1. Phases of Flight

<https://www.youtube.com/watch?time_continue=3&v=IZjLJxSYdRM>

**up\_and\_down.py**

I've included all the code from up\_and\_down.py below for your reference. Use this code as you answer the questions at the bottom of this page. Note that the backyard\_flyer project code uses the term 'state' instead of 'phase' but they refer to the same concept.

**import** time

**from** enum **import** Enum

**import** numpy **as** np

**from** udacidrone **import** Drone

**from** udacidrone.connection **import** MavlinkConnection

**from** udacidrone.messaging **import** MsgID

**class** **Phases**(Enum):

MANUAL = 0

ARMING = 1

TAKEOFF = 2

LANDING = 3

DISARMING = 4

**class** **UpAndDownFlyer**(Drone):

**def** **\_\_init\_\_**(self, connection):

super().\_\_init\_\_(connection)

self.target\_position = np.array([0.0, 0.0, 0.0])

self.in\_mission = **True**

*# initial state*

self.flight\_phase = Phases.MANUAL

*# register all your callbacks here*

self.register\_callback(MsgID.LOCAL\_POSITION,

self.local\_position\_callback)

self.register\_callback(MsgID.LOCAL\_VELOCITY,

self.velocity\_callback)

self.register\_callback(MsgID.STATE,

self.state\_callback)

**def** **local\_position\_callback**(self):

**if** self.flight\_phase == Phases.TAKEOFF:

*# coordinate conversion*

altitude = -1.0 \* self.local\_position[2]

*# check if altitude is within 95% of target*

**if** altitude > 0.95 \* self.target\_position[2]:

self.landing\_transition()

**def** **velocity\_callback**(self):

**if** self.flight\_phase == Phases.LANDING:

**if** ((self.global\_position[2] - self.global\_home[2] < 0.1) **and**

abs(self.local\_position[2]) < 0.01):

self.disarming\_transition()

**def** **state\_callback**(self):

**if** **not** self.in\_mission:

**return**

**if** self.flight\_phase == Phases.MANUAL:

self.arming\_transition()

**elif** self.flight\_phase == Phases.ARMING:

self.takeoff\_transition()

**elif** self.flight\_phase == Phases.DISARMING:

self.manual\_transition()

**def** **arming\_transition**(self):

print("arming transition")

self.take\_control()

self.arm()

*# set the current location to be the home position*

self.set\_home\_position(self.global\_position[0],

self.global\_position[1],

self.global\_position[2])

self.flight\_phase = Phases.ARMING

**def** **takeoff\_transition**(self):

print("takeoff transition")

target\_altitude = 3.0

self.target\_position[2] = target\_altitude

self.takeoff(target\_altitude)

self.flight\_phase = Phases.TAKEOFF

**def** **landing\_transition**(self):

print("landing transition")

self.land()

self.flight\_phase = Phases.LANDING

**def** **disarming\_transition**(self):

print("disarm transition")

self.disarm()

self.flight\_phase = Phases.DISARMING

**def** **manual\_transition**(self):

print("manual transition")

self.release\_control()

self.stop()

self.in\_mission = **False**

self.flight\_phase = Phases.MANUAL

**def** **start**(self):

self.start\_log("Logs", "NavLog.txt")

print("starting connection")

super().start()

self.stop\_log()

**if** \_\_name\_\_ == "\_\_main\_\_":

conn = MavlinkConnection('tcp:127.0.0.1:5760',

threaded=**False**,

PX4=**False**)

drone = UpAndDownFlyer(conn)

time.sleep(2)

drone.start()

**QUESTION 1 OF 3**

What flight phase does the drone transition to *after* Phases.DISARMING?

* Phases.MANUAL
* 

Phases.ARMING

* 

Phases.TAKEOFF

* 

Phases.LANDING

* 

None, Phases.DISARMING is the last phase.

SUBMIT

**QUESTION 2 OF 3**

Which of the following criteria must be met in order to begin a disarming\_transition?

Check all that apply.

* The drone must be in the LANDING flight phase.
* The drone must be less than 0.01 meters above ground level.
* 

The drone's (x,y)(*x*,*y*) position must be within 0.1 meters of it's original "home" position.

SUBMIT

**QUESTION 3 OF 3**

Sort the following methods into the chronological order in which they are called.

*Submit to check your answer choices!*

**ORDER METHOD IS CALLED**

**METHOD NAME**

First

self.arming\_transition()

Second

self.takeoff\_transition()

Third

self.landing\_transition()

Fourth

self.disarming\_transition()

Fifth

self.manual\_transition()

SUBMIT

1. Project Development Workflow Options

**Project Development Workflow Options**

There are **two** approaches you can take to develop and submit this project. *You will only take one of these approaches*.

**Option 1: local development (recommended)**

The first workflow option involves downloading the flying car simulator to your own computer and working locally to develop your solution to the project. You will see instructions for how to get started with this approach when you press the "Next" button in the lower right corner of the classroom.

**Option 2: Virtual Machine development**

Some students have been having difficulties getting everything setup locally. If you are unable to get local setup working then you can develop and submit your project using a virtual machine that you access through the classroom. Instructions for developing with this workflow are provided in the section **after** the local development instructions.

**Important** - If you are able to complete the local setup instructions successfully then you should ignore the VM instructions!

1. Backyard Flyer (local development)

**Backyard Flyer**

**1 - Get the Code**

1. Navigate to your FCND-Term1-Starter-Kit directory.
2. Make sure your virtual environment is activated
3. source activate fcnd
4. Clone the backyard flyer repository.
5. git clone https://github.com/udacity/FCND-Backyard-Flyer.git
6. Navigate to that repo.
7. cd FCND-Backyard-Flyer

**QUESTION 1 OF 3**

Were you able to clone the backyard flyer repository?

* Yes
* 

No

SUBMIT

**2 - Run the Solution**

In this project you will be flying the drone in a square shape. We are providing one working solution, but for your implementation you should feel free to determine your own box size and altitude. We recommend **not** looking at the code for this solution unless you really need to.

In order to run the solution you should:

1. Switch to the solution branch of the backyard flyer:
2. git checkout solution
3. Start the simulator that you downloaded earlier. If you don't have the simulator you can [**download the simulator here**](https://github.com/udacity/FCND-Simulator-Releases/releases).
4. Run backyard\_flyer.py
5. python backyard\_flyer.py

**QUESTION 2 OF 3**

Were you able to run the backyard flyer solution and see the drone fly in a square?

* 

Yes

* 

No

SUBMIT

**3 - Read the documentation!**

The drone API you'll be working with has [**thorough documentation**](https://udacity.github.io/udacidrone/docs/drone-api.html). Use it to understand the properties and methods you have access to!

**4 - Solve it Yourself!**

1. Return to the master branch:
2. git checkout master
3. Open backyard\_flyer.py in your favorite text editor. I recommend [**Sublime Text**](https://www.sublimetext.com/3).
4. Whenever you want to test your solution, run backyard\_flyer.py
5. python backyard\_flyer.py

**QUESTION 3 OF 3**

Were you able to get your drone to takeoff, fly in a square, and then land?

* Yes
* 

No

SUBMIT

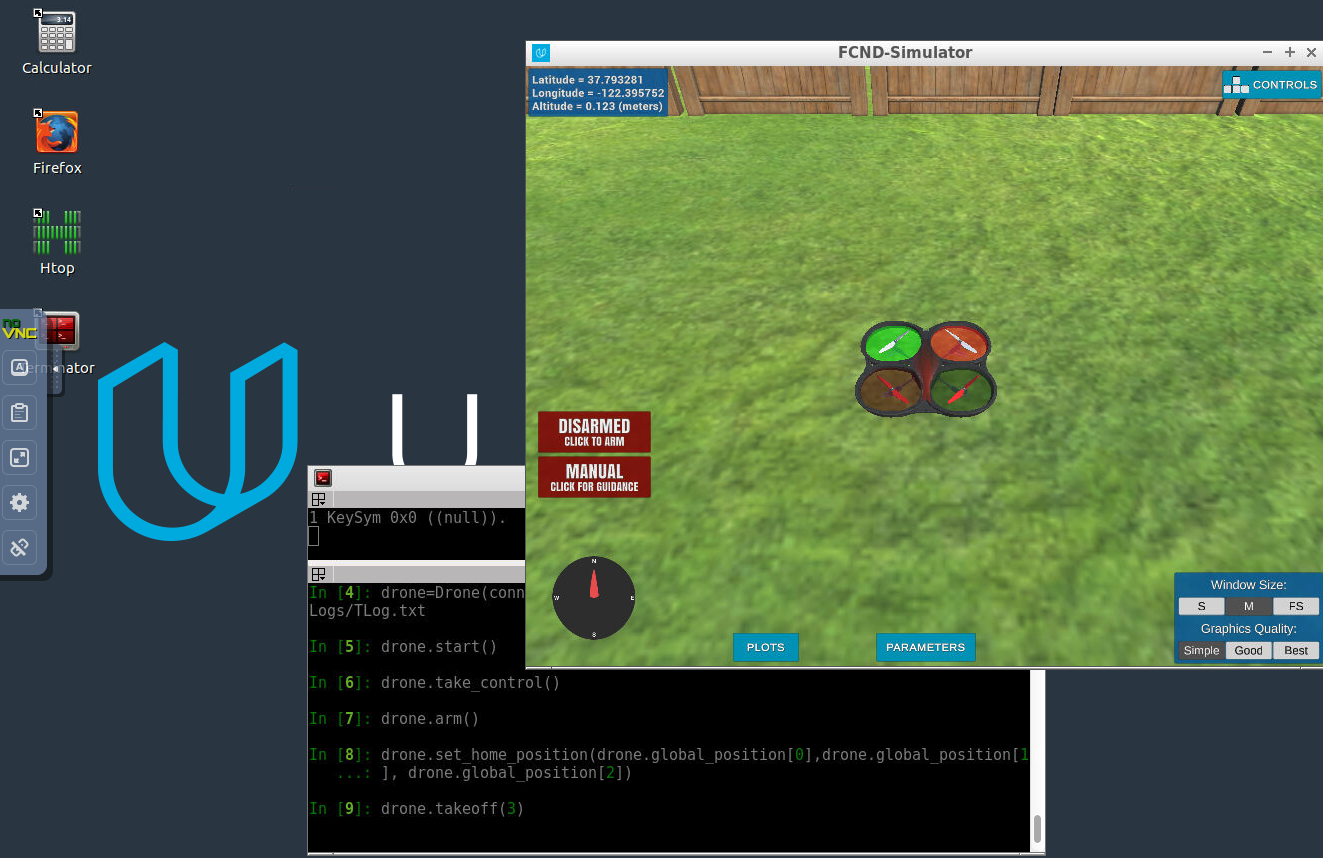
**5 - Submit!**

You'll submit your backyard\_flyer.py file on the project page.

1. Virtual Machine Intro

**Your Cloud Workspace**

You should only complete this section (and the following) if you were unable to get setup locally on the previous page.

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

**Udacity Cloud VM**

For this project, you will have the option to complete all of your work in a cloud-based GPU supported virtual machine (VM) right in the Udacity classroom! Graphics rendering in the VM seems to work best for internet connections faster than 10Mbps, so if you're on a good connection we highly recommend checking it out! You can test the speed of your connection by going to [**fast.com**](https://fast.com/). Then simply follow along with these instructions for getting started and you'll be up and running in no time with your new VM.

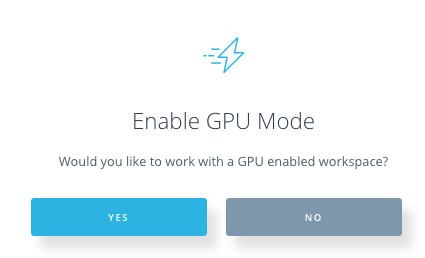
**Work on your local machine**

If you would prefer to complete the project on your local machine, then you can skip back to the [**instructions for local setup**](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/6d8be91e-0df2-4558-8723-727cf9b64243).

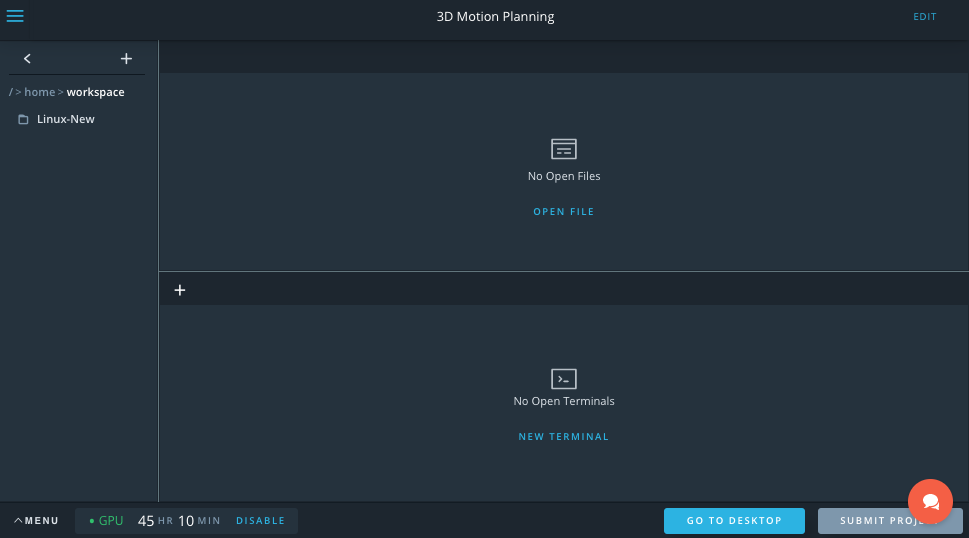
**VM Setup Instructions**

The workspace VM is already setup with Python and the simulator for the Backyard Flyer project. If you run into any problems with using the VM please file bugs, suggestions and grievances in the [**waffle board**](https://waffle.io/udacity/fcnd-issue-reports).

When you click "Next" at the bottom of this page to advance to the Backyard Flyer project in the classroom you'll first see a popup asking if you want to enable GPU mode. Choose "yes".

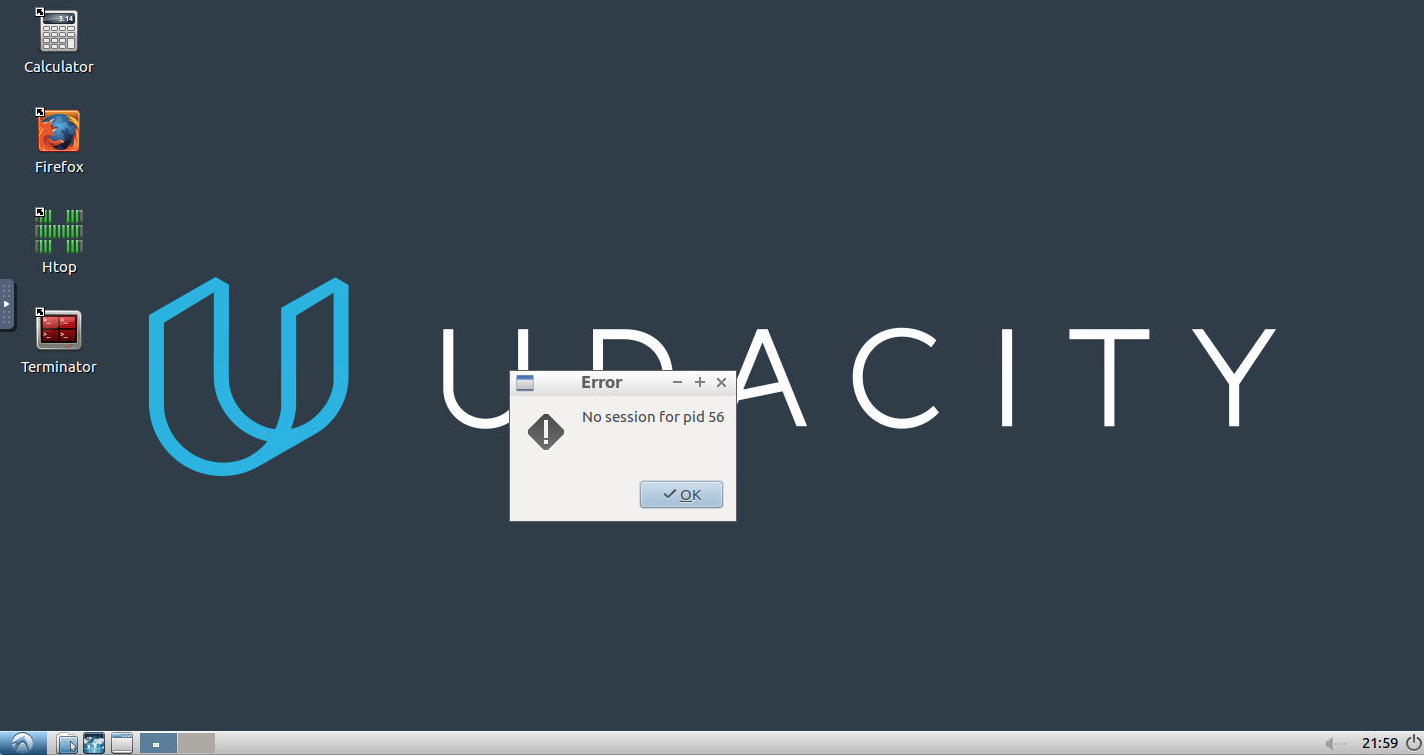
**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

Next, your workspace will be configured and you'll land in a development environment that looks like this:

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

Click the button in the lower right that says "GO TO DESKTOP" and you'll launch another browser window, where you'll land at the desktop of your Linux VM that looks like this:

(**Note: the VM currently does not work with Safari, please use Chrome or other browser**)

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

Don't worry about the error message, it's harmless, just click "OK". Next, double click on the "Terminator" icon on the desktop to open terminal. In the terminal you need to issue 3 commands:

cd /home/workspace/linux-64bit/

./FCND-Sim\_Linux\_64-bit

**How to copy/paste:**At this point you may be wondering how to copy and paste between your local machine and the VM! It's a bit cumbersome unfortunately, but here's how it works:

* copy text on local machine
* open the side-bar menu at the left side of the VM desktop
* click the clipboard icon and paste text in there
* use ctrl-shift-v to paste into terminal (or ctrl-v to paste into Firefox browser)

Reverse these steps to copy/paste from the VM to your local machine.

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

If the above method proves too cumbersome, as long as whatever you want to copy/paste is online, you can also just open up a browser (Firefox) in the VM and navigate to the material you want to copy / paste.

After issuing those commands, your simulator should launch! Now your desktop will look something like this:

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

Click the button in the simulator that says "Click to Arm" and try some manual flying to explore this new environment. Click the "Controls" button in the upper righthand corner of the sim to get a reminder of keyboard and mouse functionality in manual mode.

If all of this works, you're ready to move on to testing autonomous flying and planning in this new environment. In a new terminal, run the following commands to clone the [**project repository**](https://github.com/udacity/FCND-Backyard-Flyer) into your /home/workspace directory:

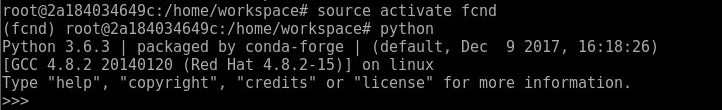
cd /home/workspace

git clone https://github.com/udacity/FCND-Backyard-Flyer

Next, activate the fcnd conda environment by typing source activate fcnd and then python at the prompt. You should now be running Python 3.6 in the fcndconda environment.

source activate fcnd

python

**[[](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/93b5488e-4641-4ec4-aecf-45763670b2b9)**

With that you have everything you need to complete this lesson and the Backyard Flyer project right in your browser. Keep your workspace desktop environment open in one tab and in another tab, head back to [**Simulator Exploration Tutorial**](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/0878b080-615e-419a-bfff-05e77f9e8fea) to get familiarized with the simulator and supporting code. Or if you're ready to complete the project, you can move on to the [**instructions for completing the project**](https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/99603a59-7bb7-4d94-9dc2-f658c29e1019).

**Submitting your project through the workspace VM**

When you're satisfied with your solution, you can submit your project directly through the VM workspace. The way this works is that when you click "Submit Project" in the workspace IDE, everything in your /home/workspace folder will be zipped up and submitted. So the first step is to move everything you do not want to submit out of the /home/workspace directory.

The one thing you absolutely must remove from /home/workspace is the simulator (with the simulator the submission would be too big and fail to go through). You can temporarily move this to /opt while you submit your project.

cd /home/workspace

mv FCND-Sim-Linux /opt

Next, move any other files or folders you do not intend to submit out of /home/workspace and then click the "Submit Project" button at the lower right in your workspace IDE (note: not in the VM desktop environment, but next to the "Go To Desktop" button on the IDE page).

When you have finished submitting, move any files you want to keep, including the simulator back into /home/workspace so they will be saved for your next visit to the VM.

1. Backyard Flyer (X-Windows)

<https://classroom.udacity.com/nanodegrees/nd787/parts/5aa0a956-4418-4a41-846f-cb7ea63349b3/modules/2c43620c-78cc-427f-afc1-e6e0d42f82f0/lessons/13c0ca8d-b11e-405a-a831-6ca38cb60246/concepts/aaa31825-33c4-43eb-9c31-2b745985d56a>

1. Project: Backyard Flyer

Project Submission

**Backyard Flyer Project Instructions**

Make a simulated drone fly in a square shape!

**Project Steps**

1. Open the **Backyard Flyer (local development)** section of this lesson and follow instructions there.
2. Make sure you check out the [**drone API documentation**](https://udacity.github.io/udacidrone/docs/drone-api.html)!
3. You will be submitting your completed version of the backyard\_flyer.py file here (by clicking the Submit button at the bottom of the page).

**Evaluation**

Once you have completed your project, use the [**Project Rubric**](https://review.udacity.com/#!/rubrics/1216/view) to review the project. If you have covered all of the points in the rubric, then you are ready to submit! If you see room for improvement in any category in which you do not meet specifications, keep working!

**Mission Code**

You should complete all sections of the BackyardFlyer class that are marked with # TODOs.

1. Complete the state transition methods for Drone class: arming\_transition, takeoff\_transition, waypoint\_transition, landing\_transition, and disarming\_transition.
2. Complete the following callbacks: state\_callback, local\_position\_callback, velocity\_callback.

The callbacks check appropriate criteria dependent on the current state and transition to the appropriate next state when those criteria are met. **Criteria cannot be based on time!**

**Mission Analysis**

Running backyard\_flyer.py correctly commands the vehicle to fly the box mission.

Your project will be evaluated by a Udacity reviewer according to the same Project Rubric. Your project must "meet specifications" in each category in order for your submission to pass.

**Submission**

When you're ready to submit, click the submit button below and upload a zip file which contains your backyard\_flyer.py file.

**Feedback**

Please fill out the [**Backyard Flyer Project Feedback Form**](https://goo.gl/forms/VQJzFUe3x1LQV6H82) after you have completed the project.

**Share Your Project**

Have something fun to share? Did your drone fly in a square, meet the fence, or say hello to a tree? We love to see any and all of your creations!

What you’ve accomplished is no small feat. Give yourself a pat on the back and some well-deserved recognition by sharing your amazing project with your network.

Create a gif or youtube video of your project implementation in the simulator and share with us in the [**Flying Car Slack Community**](https://udacity-flying-car.slack.com/messages/C77GCKZ8U) and on social media using the links below!

Make sure to use **@Udacity**, **#flyingcarnd**, **#backyardflyer**, and **#madewithudacity** in your posts!

表單的頂端

Enter the full URL of your Gif or Youtube video:  
 Share on Facebook

表單的底部

表單的頂端

 Share on LinkedIn

表單的底部

 Congratulations! You've completed this projectVIEW SUBMISSION