Table of Contents

[I. Introduction 2](#_Toc57025002)

[II. Pre-Requisites 2](#_Toc57025003)

[III. Installations 3](#_Toc57025004)

[A. IDE 3](#_Toc57025005)

[B. Brew 3](#_Toc57025006)

[C. Pipenv 3](#_Toc57025007)

[D. Python 3](#_Toc57025008)

[E. Source from Tar 4](#_Toc57025009)

[F. Source from Git (If tar provided – can skip) 4](#_Toc57025010)

[IV. Code Layout 5](#_Toc57025011)

[A. Source code 5](#_Toc57025012)

[B. Unit tests 6](#_Toc57025013)

[V. Running the Application 7](#_Toc57025014)

[VI. Tasks 8](#_Toc57025015)

[A. Task 1 : Display wind direction 8](#_Toc57025016)

[B. Task 2 : Display wind cardinal direction 11](#_Toc57025017)

[C. Task 3 : Unit Tests 13](#_Toc57025018)

[D. Task 4 : Display weather from Longitude and Latitude 16](#_Toc57025019)

[VII. Documentation references 19](#_Toc57025020)

# Introduction

This workshop will guide you through the process of creating a simple **python3** web-app, utilizing the **Flask** web framework. The app will retrieve weather information for a given location or longitude/latitude and displays max/min/current temperature, wind speed information and a brief weather status along with a .png icon for display.

**OpenWeatherMap** is one of the most popular and common weather APIs, we will be using this in conjunction with **Flask** (a popular python-based web-framework) to create our python3 application.

This workshop will guide you through the following steps:

* Installing an IDE for python3 development
* Installing brew and pipenv dependencies
* Cloning the project repository
* Initialising pipenv with python dependencies
* Setting up the config file with API key for **OpenWeatherMap**
* Running Flask and the initial webapp
* Extending functionality to include wind speed per location
* Unit-testing our config validation method
* EXTENSION: Expanding the app to search by longitude/latitude

# Pre-Requisites

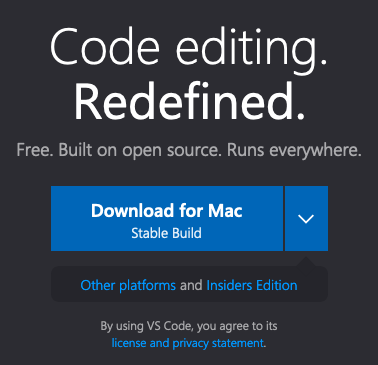
* OpenWeatherMap valid API key
* Macbook or linux-based machine with a terminal emulator

# Installations

## IDE

Firstly, we will require an **IDE** (integrated development environment) to speed up the development process and provide useful helpers when writing Python3 code. **VSCode** is recommended for its simplicity and lightweight installation, this can be installed from here: <https://code.visualstudio.com/>

Follow the download link and unzip the .zip archive, copy the application to your “Applications” folder.



## Brew

Brew is an extremely useful and versatile package manager tool for MacOS (like apt for Ubuntu or yum for Fedora) and is necessary to install the next dependency.

Run the following command to install Brew:

/bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/master/install.sh>

## Pipenv

When developing an application in Python, we often rely on packages ([https://pypi.org/**)**](https://pypi.org/)%20) to speed up the development of our code. For this particular project, we require the package **pipenv** to allow the creation of a virtual environment containing all dependencies.

Pipenv can be installed as follows:

brew install pipenv

## Python

**POTENTIAL SECTION HERE ON INSTALLING PYTHON3, VERIFY IF MACOS CATALINA SHIPS WITH IT IN** /usr/bin/python3, if not we can install with brew update && brew upgrade python

## Source from Tar

You will be provided with a .tar.gz containing all the workshop content. Untar this to your local filesystem.

## Source from Git (If tar provided – can skip)

We will need to clone the workshop source code from a git repository, try run **“git”** from a terminal. A prompt to install “xcode developer tools” will appear and install git. If no prompt appears – great! You already have git installed on your machine.

# Code Layout

## Source code

The code is organized into a typical project structure as follows:

A close up of text on a black background

Description automatically generated

1. **Pipfile & Pipfile.lock**: this file contains all the dependencies for the current project, the lock file ensures versions aren’t automatically updated.
2. **README.md**: most projects contain a README file with basic run instructions and useful information.
3. **instance/ :** this is a folder automatically generated by **pipenv**.
4. **tests/** : all unit tests for the code are within this directory.
5. **weather/** : all main application code is within this directory.
6. **weather/config.yaml** : This file contains the following configuration parameters:
   1. API key: this is required to interact with **OpenWeatherMap** API

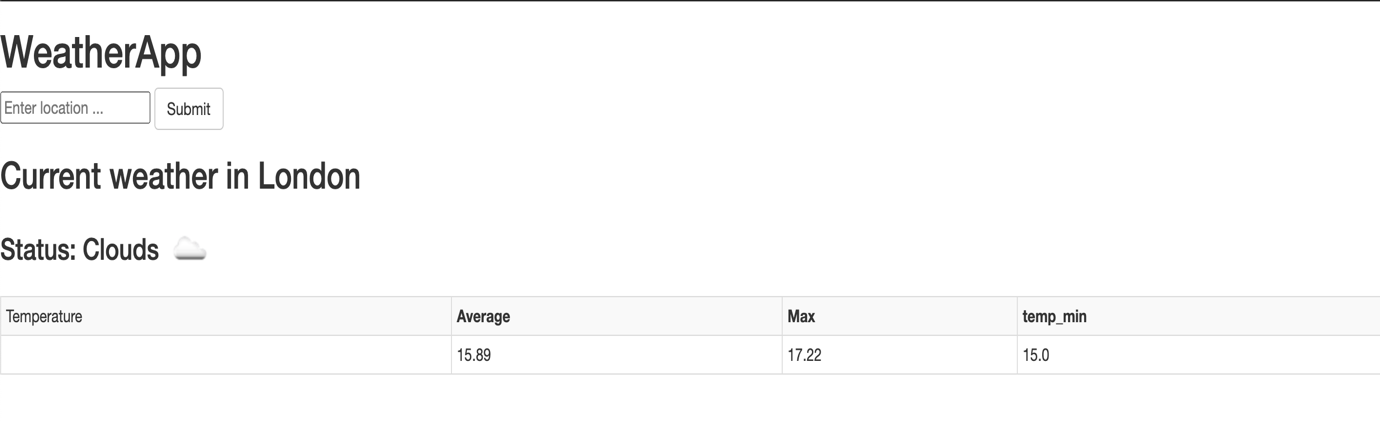
Unit in which we want to measure temperature, currently configured as ‘**celsius’** but can also be ‘**kelvin’** or ‘**fahrenheit’**

1. **weather/loader**: This module loads and validates the **config.yaml** file ensuring both **api\_key** and a valid value for **temperature** are set.
2. **weather/weather\_api** : This is module creates a wrapper around **pyowm**.

**Pyowm** is an open source module we are using as a dependency within this project, it allows for simple interaction and integration with **OpenWeatherMap** API methods.

Further details about this package can be found here <https://github.com/csparpa/pyowm>

1. **weather/templates**: This folder contains html templates that are used by flask to render webpages that can directly interact with python variables (input a location, output some information).



## Unit tests

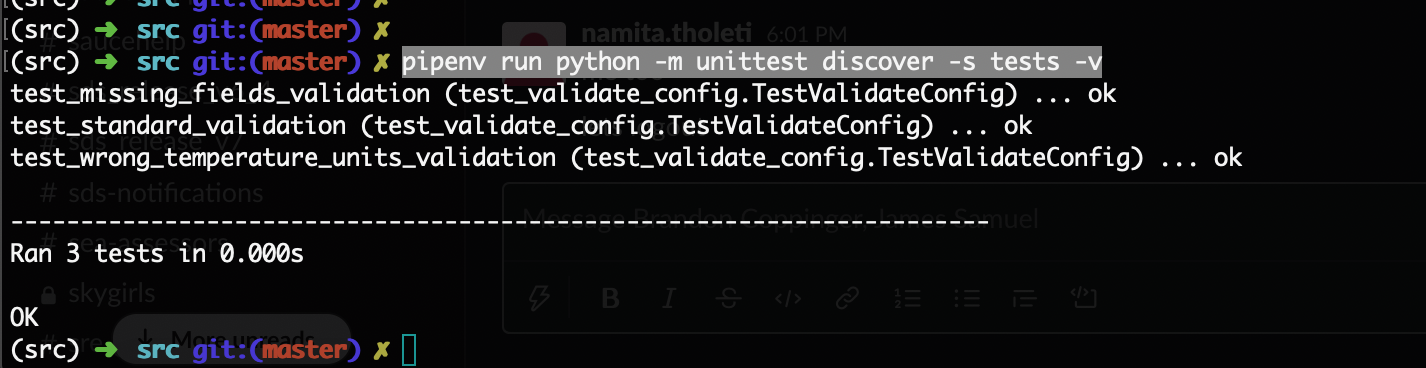
The unit test folder is where currently there is a unittest file test\_validate\_config.py to verify the config loader functionality.

The unit tests are currently blank and will need to implemented as part of your task.

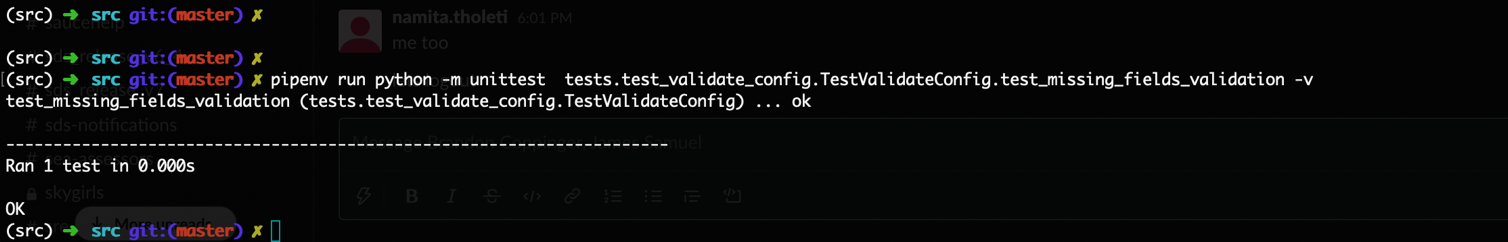
Running the unit tests:

Tests can be run using the following command:

pipenv run python -m unittest discover -s tests -v



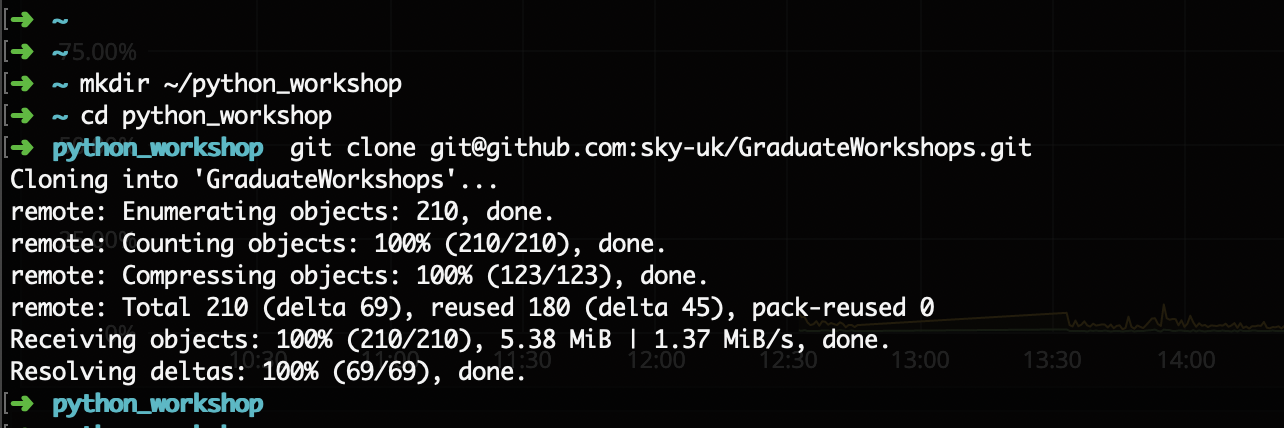
You can target a specific test by referencing the function name within the “**TestValidateConfig**” class.



Note: in case you get any errors related to ‘**ModuleNotFoundError’** then it is worth trying to run ‘**pipenv sync’** to ensure the dependencies are locked to the correct version.

# Running the Application

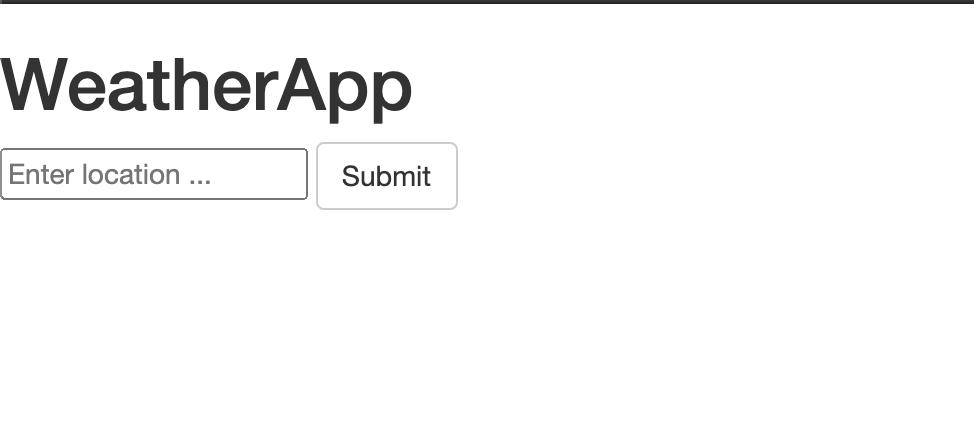
1. Within the terminal, make a new directory for the project in your home folder called “**python\_workshop**” and navigate to that directory:

****

1. Place the API key for **OpenWeatherMap** you received as part of this workshop into **“src/weather/config.yaml”** under “**api\_key”.**
2. Now run the following commands

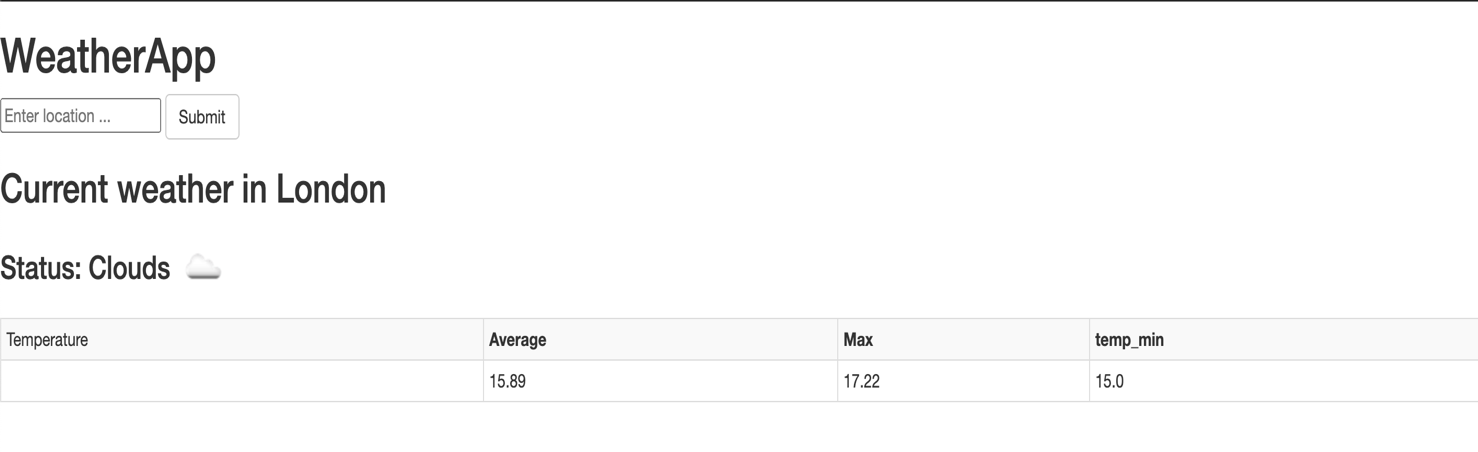
***./run.sh*** *NOTE: This will execute the following commands, and has been included as a helper script:  
`FLASK\_APP=weather/main.py pipenv run python -m flask run`*

1. Success! You have a development webserver now running locally serving your python web application! Click on the URL <http://127.0.0.1:5000/> and it should take you to the web app as follows:

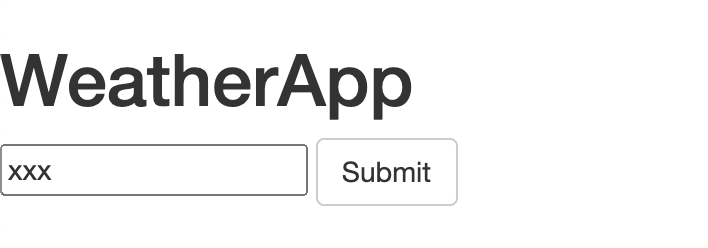


1. Now enter a location such as ‘London’, and click submit. The app will then extract a variable from this request form, and send via an argument to **get\_current\_weather\_at\_location()**.

This will in turn retrieve a dictionary object which is, rendered via Flask onto this webpage. (this can be observed in “**templates/index.html**”, using jinja syntax to access elements of a dictionary) - <https://jinja.palletsprojects.com/en/2.11.x/templates/>



1. If you enter an incorrect location say ‘xxx’ and submit then an error page will be displayed. This is handled by checking for any errors returned by our weather\_api in the form of a dictionary. “**templates/500.html**” renders any errors passed as arguments.





# Tasks

## Task 1 : Display wind direction

### Get wind info

We query the **OpenWeatherMap** **API** for weather information using the **weather\_at\_place** method within **pyowm**. An example of the structured API response that we receive is as follows:

{"coord": { "lon": 139,"lat": 35},

"weather": [

{

"id": 800,

"main": "Clear",

"description": "clear sky",

"icon": "01n"

}

],

"base": "stations",

"main": {

"temp": 281.52,

"feels\_like": 278.99,

"temp\_min": 280.15,

"temp\_max": 283.71,

"pressure": 1016,

"humidity": 93

},

"wind": {

"speed": 0.47,

"deg": 107.538

},

"clouds": {

"all": 2

},

"dt": 1560350192,

"sys": {

"type": 3,

"id": 2019346,

"message": 0.0065,

"country": "JP",

"sunrise": 1560281377,

"sunset": 1560333478

},

"timezone": 32400,

"id": 1851632,

"name": "Shuzenji",

"cod": 200

}

This response contains a dictionary with wind info which contains the two keys “**speed**” and “**deg**”.

In **api.py**, under the **weather\_api** folder, we have a function called **get\_current\_weather\_at\_location().** Here we retrieve a **weather\_location** object.

Within the “**weather\_api/api.py**” method, note the **get\_status()** and **get\_temperature()** functions. These are helper methods within **pyowm** we can use to retrieve useful pythonic objects from the **OpenWeatherMap** API.  
Using the docs below as a module reference, implement an existing method within **pyowm** to retrieve wind speed in degrees.

<https://buildmedia.readthedocs.org/media/pdf/pyowm/latest/pyowm.pdf>

*HINT: This is a method within* ***weather\_at\_place().***

*Note:* ***temperature\_unit*** *is defined in the* ***config.yaml*** *file, it is recommended to define and use* ***wind\_unit*** *here as the* ***unit*** *argument to* ***get\_wind()****.*

*Using config.yaml to store these options allows our application to be highly configurable, and we desire a single source of truth for all options.*

### Display wind information on the webpage

Once we have the wind info it should be appended to the dictionary **weather\_dict** in the **get\_current\_weather\_at\_location()** function. This can be done in the same way as temperature.

**weather\_dict** then needs to be referenced in **templates/index.html**. Reference the wind speed and degree direction in index.html as follows.

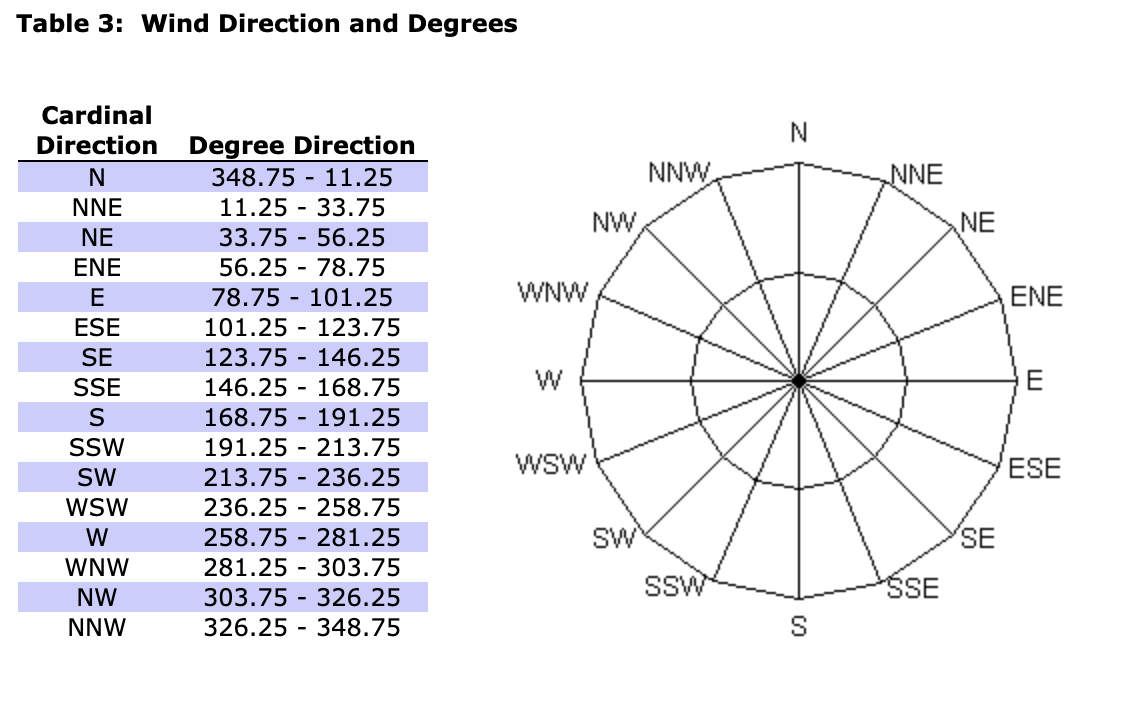
*Note: Ensure the keys in your dictionary match the value you are querying via the below Jinja syntax.*

<h4>Wind speed: {{ weather['speed'] }}</h4>  
<h4>Wind degree direction: {{ weather['deg'] }}</h4>

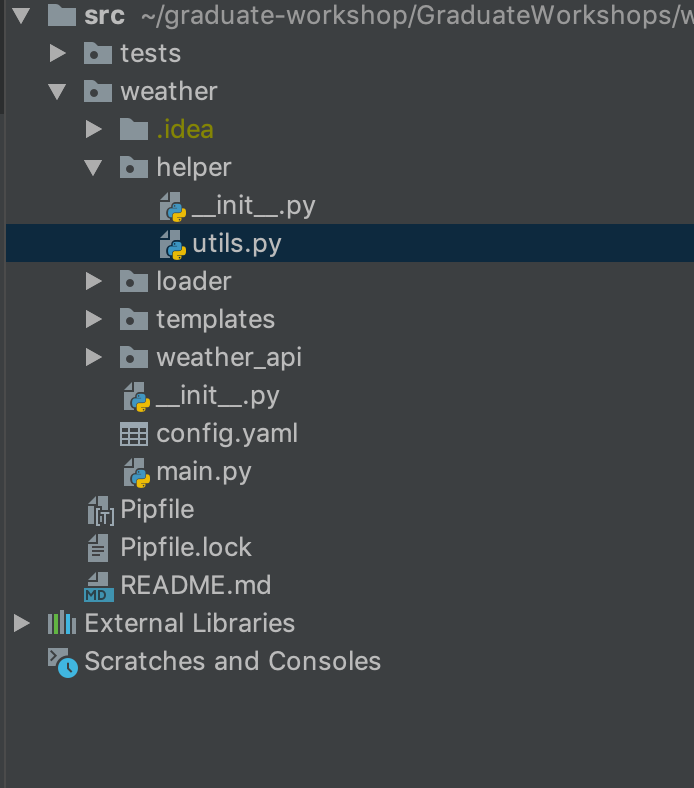
## Task 2 : Display wind cardinal direction

### Wind degrees to text utility function

The following table can be used to map the wind degree direction to wind cardinal direction. In this task we will only consider N, NE, E, SE, S, SW, W, NW directions.



To convert wind degrees to a cardinal direction we will first need to create a helper function. Under the weather folder create a folder titled **“helper”**, then create a file titled **utils.py.**



In this file we need to add the function **wind\_deg\_to\_text()** as given below.

# Accepts wind direction in degrees  
def wind\_deg\_to\_text(degree):  
 sectors = ['Northerly', 'North Easterly', 'Easterly', 'South Easterly', 'Southerly', 'South Westerly', 'Westerly',  
 'North Westerly']  
  
 degree += 22.5  
  
 if degree < 0:  
 degree = 360 - abs(degree) % 360  
 else:  
 degree = degree % 360  
  
 which\_sector = int(degree // 45)  
 return sectors[which\_sector]

*This function will calculate the wind degree direction using the table illustrated above, and map to the corresponding cardinal direction string within the “sectors” dictionary.*

### Referencing the utility function

Now we are successfully retrieving the wind speed in degrees from **OpenWeatherMap**, it would be useful to utilise the helper function we created above to map this to a friendly cardinal direction.

The above created utility function should be referenced within the **get\_current\_weather\_at\_location()** function in “**weather\_api/api.py”**.

Pass wind speed in degrees to the utility function **wind\_deg\_to\_text**() and return the cardinal direction as as string.

### Display wind cardinal direction

Once cardinal direction is returned as a string, store it within the weather\_dict dictionary in **get\_current\_weather\_at\_location**().

**“templates/index.html”** will need to be modified to display this newly created dictionary key/value. Add a html line to display the wind direction referencing it from this dictionary.

Example:

<h4>Wind cardinal direction: {{ weather['wind\_direction] }}</h4>

*NOTE: This must be passed as an argument to flask’s render\_template function to be used within jinja templating.*

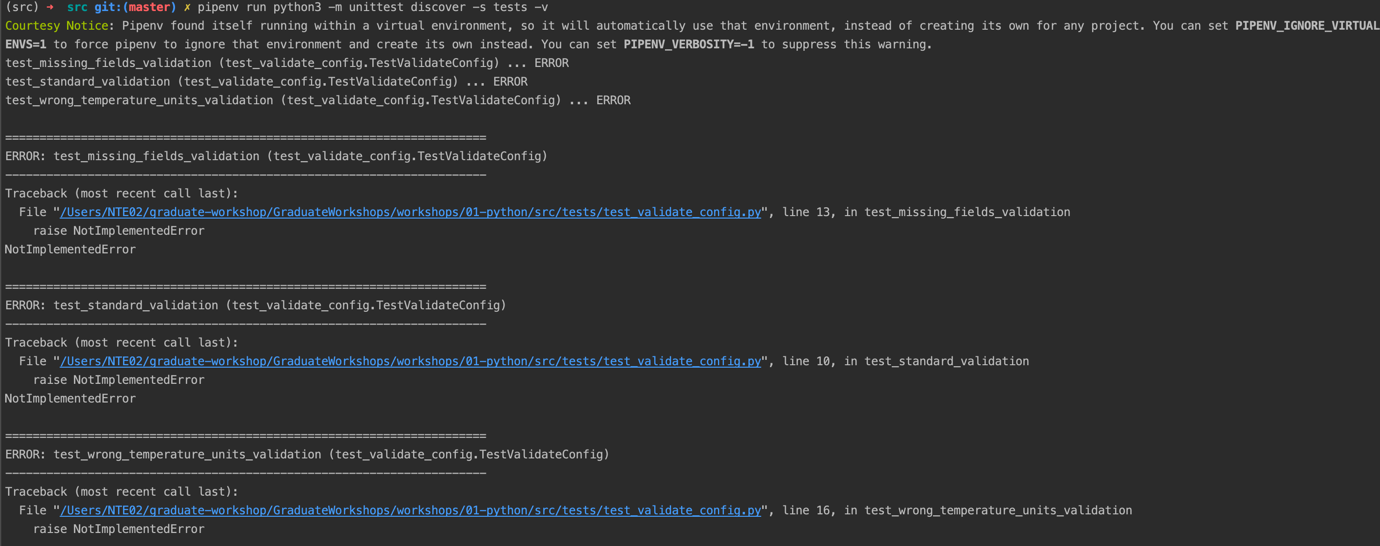
## Task 3 : Unit Tests

### To implement test\_validate\_config.py

This file is mainly to test validate\_config functionality. Currently the functions in the file are not implemented and the template is s follows

class TestValidateConfig(unittest.TestCase):  
 *"""Unit tests to ensure validate\_config works correctly"""* # *TODO fill in unit tests* def test\_standard\_validation(self):  
 raise NotImplementedError  
  
 def test\_missing\_fields\_validation(self):  
 raise NotImplementedError  
  
 def test\_wrong\_temperature\_units\_validation(self):  
 raise NotImplementedError

As you can see in the file template, the aim of a unit test is to cover the normal working scenario, the failure scenarios and any edge cases. If you run the unit test now then they will error



Each of these unit tests must be implemented. If we take an example of the first case then we need to call validate\_config with a correct set of values and then test the behaviour. We also need to make sure that the return value is what we expect, in the example below we are using assertTrue to achieve this. The various unittest assertions can be found here - <https://docs.python.org/3/library/unittest.html>

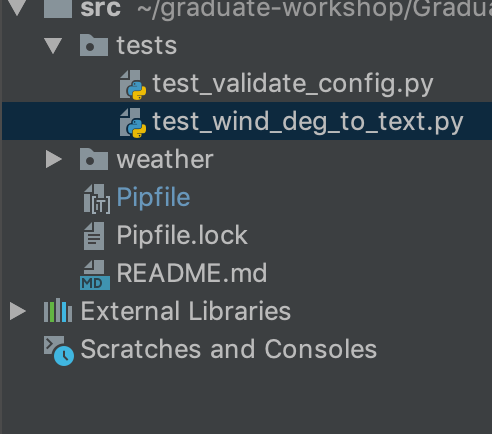
def test\_standard\_validation(self):  
 config = {"temperature\_unit": "celsius", 'api\_key': "xxx"}  
 self.assertTrue(validate\_config(config))

### For Wind degrees to text added in Task 2

In Task2 we created the function **wind\_deg\_to\_text()** within “**helper/utils.py”**.

Unit tests are now required to ensure that that the function behaves correctly when given different input parameters..

In the project we already have a tests folder, you can create a new file in there as follows:



In this file we can use the following template to for our unit tests:

import unittest  
from weather.helper.utils import wind\_deg\_to\_text  
  
class TestWindDegToText(unittest.TestCase):  
  
   
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Within the **TestWindDegToTxt** class we now need to add some positive and negative test cases.

1. Positive test case: If the function **wind\_deg\_to\_text**() receives an input value between 0 and 360 then it will be a positive test case. It can be something as follows:

def test\_standard\_wind\_deg(self):  
 wind\_direction = wind\_deg\_to\_text(60)  
 self.assertTrue(wind\_direction == 'North Easterly')

In the above scenario we have given a normal wind degree and are checking if the return value is as we expect – the cardinal direction is mapped correctly.

1. Negative test case: If the function **wind\_deg\_to\_text**() recieves an input value which is negative. e.g: -45 or something greater than 360.  
   **Write a test case for this scenario.**
2. Edge cases: If the function **wind\_deg\_to\_text**() receives an input value which is 0 or 360.

**Write a test case for this scenario.***HINT: When considering wind direction in degrees, examine 0 and 360. Refer to the cardinal direction table if required.*

Once all the test cases are in place you should be able to run the tests.

Refer to section “**B. Unit Tests:** Running the unit tests” for instructions on how to run these tests.

## Task 4 : Display weather from Longitude and Latitude

### Create Longitude and Latitude input parameters

Currently, only location is accepted as a valid input to our weather API functions. We now need to extend this input form to accommodate for Latitude and Longitude as input values.

For this we need to modify the “**templates/index.html”**. In here you will see we already have an <input> tag for location. This needs to be extended for Longitude and Latitude:

<form method='POST' action='/'>  
 <input type="text" name="location" placeholder="Enter location ..." style="max-width: 10000px;" autofocus required/>  
 <input type="text" name="longitude" id="longitude" placeholder="Enter longitude ..." style="max-width: 10000px;" autofocus required/>  
 <input type="text" name="latitude" id="latitude" placeholder="Enter latitude ..." style="max-width: 10000px;" autofocus required/>  
 <!-- -->  
 <button type="submit" class="btn btn-default">Submit</button>  
</form>

The default page when loaded should be displayed as follows:



### Read Longitude and Latitude parameters

In the “**weather/main.py**” we need to read the parameters exposed by our input form. We have a section where the location form is handled.

You can add a similar section under it to read Longitude and Latitude:

try:  
 latitude = int(request.form['latitude'])  
 longitude = int(request.form['longitude'])  
except:  
 raise Exception("Unable to get longitude or latitude in request, something has gone badly wrong!")  
 exit(1)

### Query weather API based on Longitude and Latitude

In the “**weather\_api/api.py**” create a helper function **get\_current\_weather\_at\_coordinates**() that will help us get weather based on co-ordinates.

def get\_current\_weather\_at\_coordinates(self, longtitude, latitude, unit)

"""  
Gets current weather status  
returns a dict of form   
{'status': 'Clouds', 'icon\_url': 'http://openweathermap.org/img/w/04d.png', 'temp': 14.32, 'temp\_max': 15.0,  
 'temp\_min': 12.78, 'temp\_kf': None}  
"""

You can use the function **weather\_at\_coords()** within **pyowm** to achieve this. Ensure you handle errors in an efficient way – refer to the other methods for examples.

Ensure you retrieve **status, temperature** and **icon\_url** from this object, and assemble a dictionary.

def weather\_at\_coords(self, lat, lon):  
 *"""  
 Queries the OWM Weather API for the currently observed weather at the  
 specified geographic (eg: 51.503614, -0.107331).* ***:param*** *lat: the location's latitude, must be between -90.0 and 90.0* ***:type*** *lat: int/float* ***:param*** *lon: the location's longitude, must be between -180.0 and 180.0* ***:type*** *lon: int/float* ***:returns****: an \*Observation\* instance or ``None`` if no weather data is  
 available* ***:raises****: \*ParseResponseException\* when OWM Weather API responses' data  
 cannot be parsed or \*APICallException\* when OWM Weather API can not be  
 reached  
 """*

*NOTE: ensure to handle the* ***pyowm*** *client using* ***self*** *– examples of this can be seen within* ***get\_current\_weather\_at\_location().***

### Display weather information at coordinates

Within “**templates/index.html**”, the above dictionary returned by **get\_current\_weather\_at\_coordinates**() must be displayed in a tabular format.

In “**weather/main.py**” we need to utilize **get\_current\_weather\_at\_coordinates**() and create a new dictionary variable we are able to pass to **index.html**.

Create the **coord\_weather** dictionary as follows, we must also pass this as an argument to our **render\_template** function. This ensures the variables can be accessed in html rendering.

coord\_weather = weather\_api\_wrapper.get\_current\_weather\_at\_coordinates(longitude, latitude, options['temperature\_unit'])

return render\_template("index.html", weather=weather, coord\_weather=coord\_weather, location=location, longitude=longitude, latitude=latitude)

Within “**templates/index.html**” you can use the following block to display the co-ordinate based weather info:

{% if coord\_weather %}  
 <h2>Current weather at Longitude {{ longitude }} and Latitude {{ latitude }}</h2>  
 <h3>Status: {{ coord\_weather['status'] }} <img src="{{ coord\_weather['icon\_url'] }}"></h3>  
 <table class="table table-striped table-bordered table-condensed">  
 <tbody>  
 <tr>  
 <td>Temperature</td>  
 <td><b>Average<b></td>  
 <td><b>Max<b></td>  
 <td><b>temp\_min<b></td>  
 </tr>  
 <tr>  
 <td></td>  
 <td>{{ coord\_weather['temp'] }}</td>  
 <td>{{ coord\_weather['temp\_max'] }}</td>  
 <td>{{ coord\_weather['temp\_min'] }}</td>  
 </tr>  
 <tr>  
 </tr>  
 </tbody>  
 </table>  
{% endif %}

Have a go at running the application, you should now be able to query weather information based on co-ordinates and location names!

# Documentation references

**Flask** - <https://flask.palletsprojects.com/en/1.1.x/>

**Jinja** - <https://jinja.palletsprojects.com/en/2.11.x/>

**Pyowm module reference -** <https://buildmedia.readthedocs.org/media/pdf/pyowm/latest/pyowm.pdf>

**OpenWeatherMap API reference -** <https://openweathermap.org/current>