6.1.4

Generate Random Latitudes and Longitudes

Latitude and longitude—got it. Now it's time to code. You know you want to generate enough coordinates that you'll have a good covering of common travel destinations, so you'll need coordinates that are close to a city.

For the development phase of the project, you and Jack decide that about 500 cities should be enough. Of course, if the algorithm you're about to write works, you'll be able to use and reuse it to get many more cities.

Before we write the algorithm to generate the latitudes and longitudes, we need to refresh our memory of where people live in the world. Time for another quick geography lesson!

Earth's surface is covered by 70% water while the rest is covered by land. So, we can assume 70% of the latitude and longitude coordinates we generate are positioned over a body of water, whether an ocean, major lake (e.g., Lake Superior), or major river (e.g., Amazon). Geographic coordinates over a body of water may not be close to a city, especially if in the middle of an ocean.

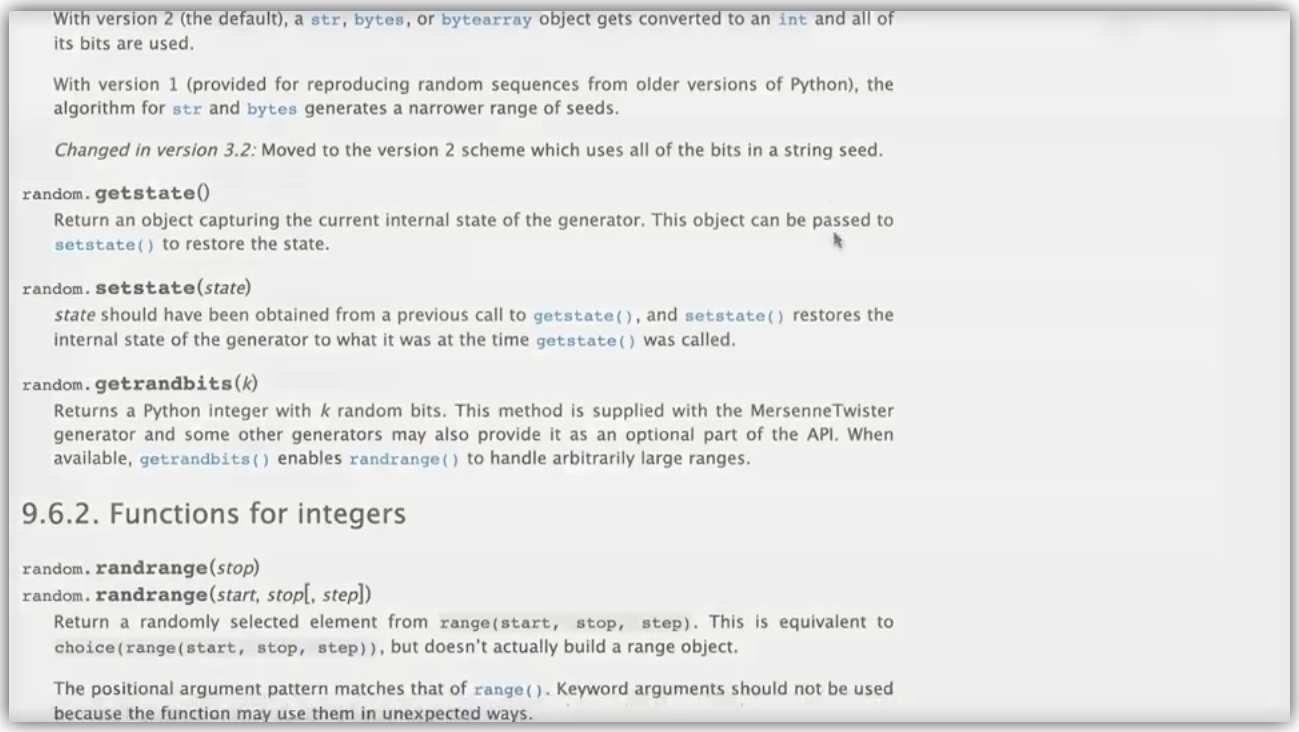
Seven continental landmasses comprise 30% of Earth's surface. Some land is uninhabitable or sparsely populated due to extreme terrain and climes (e.g., Sahara, Siberia, the Himalayas, and areas of the western United States).

First consider the bodies of water. Start with at least 1,500 latitudes and longitudes, because 500 divided by 0.3 (30% land mass) equals 1,666 latitudes and longitudes.

We'll generate random latitudes and longitudes to ensure coordinates are fairly distributed around the world. An algorithm will pick random numbers between the low and high values for latitudes and longitudes. Also, the latitudes and longitudes must be floating-point decimal numbers, as each angular unit of degrees, minutes, and seconds can be represented by a decimal number. For example, Kailua-Kona, Hawaii has the angular coordinates 19° 38' 23.9784" north and 155° 59' 48.9588" west and can be written as a decimal number as follows: 19.639994, -155.996933.

To generate random numbers, we can use the Python `random` module. This module is part of the Python and Anaconda installation, so we don't need to install it. Let's test some `random` module functions to find one that can help us.

The random Module



With version 2 (the default), a `str`, `bytes`, or `bytearray` object gets converted to an `int` and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for `str` and `bytes` generates a narrower range of seeds.

Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

`random.`**`getstate`**`()`
Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

`random.`**`setstate`**`(`*state*`)`
state should have been obtained from a previous call to `getstate()`, and `setstate()` restores the internal state of the generator to what it was at the time `getstate()` was called.

`random.`**`getrandbits`**`(`*k*`)`
Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, `getrandbits()` enables `randrange()` to handle arbitrarily large ranges.

9.6.2. Functions for integers

`random.`**`randrange`**`(`*stop*`)`
`random.`**`randrange`**`(`*start*, *stop*[, *step*]`)`
Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

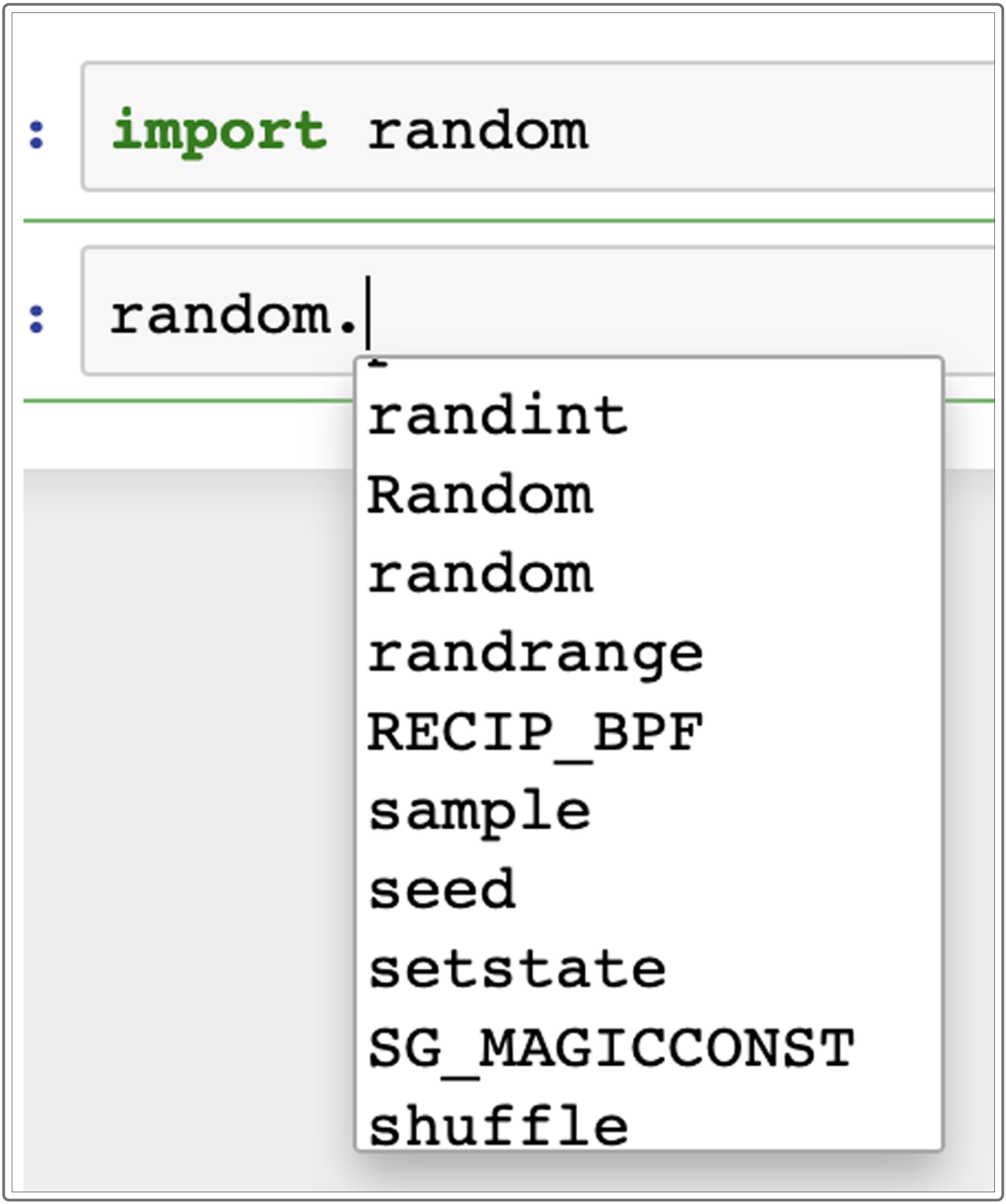
The positional argument pattern matches that of `range()`. Keyword arguments should not be used because the function may use them in unexpected ways.

First, create a new Jupyter Notebook file named `random_numbers.ipynb`.

In the first cell, import the `random` module and run the cell.

```
# Import the random module.
import random
```

In the next cell, type `random.`, and after the period, press the Tab key for a list of available `random` module functions.



For testing, we'll use the `randint()`, `random()`, `randrange()`, and `uniform()` functions.

The randint() Function

`randint` is short for "random integer." In the second cell, after `random.`, type `randint(-90, 90)`, as shown below.

```
random.randint(-90, 90)
```

When we run this cell, we'll get a single integer between -90 and 90 because we need two latitudes between -90 and 90.

This is useful information, but it doesn't get the job done for us. Remember, we need 1,500 random decimal numbers. This function will only return one integer, not a floating-point decimal, between the given intervals. Let's try the `random()` function.

The random() Function

Using the `random()` function, we can get a single floating-point decimal number between 0 and 1.0.

Add `random.random()` to a new cell and run the cell. Your output should be a decimal point number between 0 and 1.0, as shown below.

```
random.random()
```

```
0.45504101129371866
```

The `random()` function may help us. This function returns only a floating-point decimal number between 0 and 1.0. If we combine `random.randint(-90, 89)` and `random.random()` to generate a floating-point decimal between -90 and 90, we can generate a random latitude. We changed the lower range of the `randint()` because we want whole numbers up to 89, so when we add the floating-point decimal number, we'll generate latitudes between −89.99999 and 89.99999.

```
random_number = random.randint(-90, 89) + random.random()
random_number
```

```
-32.84922525804476
```

Using these two functions, we can write an algorithm that will generate latitudes between -90 and 89. Here is a small sample of what it might take to generate ten random floating-point decimal latitudes between -90 and 89.

```
x = 1
latitudes = []
while x < 11:
    random_lat = random.randint(-90, 89) + random.random()
    latitudes.append(random_lat)
    x += 1
```

In the code block above, we:

1. Assign the variable x to 1.
2. Initialize an empty list, `latitudes`.
3. We create a while loop where we generate a random latitude and add it to the list.
4. After the random latitude is added to the list we add one to the variable "x".
5. The while loop condition is checked again and will continue to run as long as x is less than 11.

The output from running this code might look like this:

```
latitudes
```

```
[-25.96953778997501,
 29.94130957005215,
 86.11841276789062,
 -82.23532680413341,
 -39.46468867602236,
 62.25071890226407,
 -89.32539284874133,
 54.87228791281589,
 -69.62327893418491,
 -65.45109625694677]
```

Next, we would have to use a similar method to get random longitudes between –180 and 180, which we can then pair with the latitudes. This looks promising, but the code to generate the latitudes above is a little long.

Let's try another function, the `randrange()` function.

The randrange() Function

The `randrange()` function behaves differently than the previous two functions. Inside the parentheses, we need to add two numbers, a lower and upper limit, separated by a comma.

For the `randrange()` function, there is an option to add a `step` parameter and set it equal to an integer, which will generate increments of a given integer value, from the lower to the upper limit.

For example, add `random.randrange(-90, 90, step=1)` to a new cell and run the cell. The output is a number between -90 and 90, where the step is the difference between each number in the sequence.

```
random.randrange(-90, 90, step=1)
```

```
-77
```

Now add `random.randrange(-90, 90, step=3)` to a new cell and run the cell. The output is a number between -90 and 90, where the difference between each number in the sequence is 3.

```
random.randrange(-90, 90, step=3)
```

```
72
```

NOTE

If you don't add the step parameter, the output will be a number with an increment of 1, which is the default integer value.

This function might help us by combining the `random.randrange()` and `random.random()` functions to generate a floating-point decimal between -90 and 90, like we did with the `random.randint()` and `random.random()` functions.

Let's look at one last function, the `uniform()` function.

The uniform() Function

The `uniform()` function will allow us to generate a floating-point decimal number between two given numbers inside the parentheses.

Add `random.uniform(-90, 90)` to a new cell and run the cell. The output should look like the following:

```
random.uniform(-90, 90)
```

```
-16.992740452265 65
```

The `uniform()` function could prove to be quite useful because it will return a floating-point decimal number! The table below reviews the functions' outputs and limitations:

Function	Output	Limitation
`randint(-90, 89)`	Returns an integer between the interval, -90 and up to 89.	Will not generate a floating-point decimal number.
`random()`	Returns a floating-point decimal number between 0 and 1.	Will not generate a whole integer.
`randrange(-90, 90, step=1)`	Returns a whole integer between the interval, -90 and 90 where the step is the difference between each number in the sequence.	Will not generate a floating-point decimal number.
`uniform(-90, 90)`	Returns a floating-point decimal number between the interval, -90 and 90.	Will not generate a whole integer.

Remember, we need to get more than a thousand latitudes and longitudes, and running one of these functions using a while loop or other methods may take more programming than needed.

To help us generate the 1500 latitudes and longitudes, we can combine the NumPy module with one of the random module functions.

The NumPy and random Modules

One way to generate more than a thousand latitudes and longitudes is to chain the NumPy module to the random module to create an array of latitudes or longitudes between the lowest and highest values, or -90° and 90°, and –180° and 180°, respectively. To accomplish this, we'll use the `uniform()` function from the random module.

REWIND

Recall that the NumPy module is a numerical mathematics library that can be used to make arrays or matrices of numbers.

Let's import the NumPy module in a new cell and run the cell.

```
# Import the NumPy module.
import numpy as np
```

NOTE

The NumPy module has a built-in random module, and supplements the built-in Python random module. There is no need to import the random module if we import the NumPy module, as it's redundant.

In the next cell add `np.random.uniform(-90.000, 90.000)` to generate a floating-point decimal number between -90.000 and 90.000. Adding the zeros past the decimal places is optional.

```
np.random.uniform(-90.000, 90.000)
```

```
44.43699159272211
```

When we use the NumPy module with the `random.uniform()` function, the parenthetical parameters contain a lower boundary (low value) and an upper boundary (high value) that are floating-point decimal numbers.

NOTE

Another option is to write the parameters as `np.random.uniform(low=-90, high=90)`.

To generate more than one floating-point decimal number between -90 and 90, we can add the `size` parameter when we use the NumPy module and set that equal to any whole number.

To see how this works, add the code `np.random.uniform(-90.000, 90.000, size=50)` to a new cell and run the cell. The output is an array of 50 floating-point decimal numbers between -90.000 and 90.000.

```
np.random.uniform(-90.000, 90.000, size=50)
```

```
array([ 17.96687837,  14.36195784,  69.32809477, -55.49345622,
         1.75422576,  43.05565827, -71.77162584, -14.42567115,
        51.79523949,  43.23458966, -41.37785462, -23.62406857,
       -13.37855367,  -6.8691874 , -50.84192853,  87.28360625,
       -60.18972261,  73.24902693,  41.48800538, -63.44483297,
        73.32733193, -68.74690001,  70.15217544, -55.02030612,
        50.58837816, -11.19406658,  61.59836396,  38.76352194,
        34.70962514,  82.55273652,  30.97675505, -47.36772117,
        10.59141067, -17.46764942, -26.84302643,  12.80951266,
        -4.06464157, -72.06059609,   3.29231855, -71.50703124,
        87.93813796, -81.63540186,  72.92436787,  15.4066607 ,
        55.84949676, -74.57043875,  74.93885592, -39.98419312,
        75.70113548, -25.56786225])
```

Now we are getting somewhere—all we need to do is increase the parameter size to 1,500.

Is this method faster than creating a while loop like we did before? Let's test this for a size of 1,500.

To test how long a piece of code or function takes to run, we can import the "timeit" module and use the `%timeit` magic command when we run our code or call the function.

First, import the timeit module in a new cell, and run the cell.

```
# Import timeit.
import timeit
```

Next, add the `%timeit` magic command before the `np.random.uniform(-90.000, 90.000, size=1500)` in a new cell. The cell should look like this:

```
%timeit np.random.uniform(-90.000, 90.000, size=1500)
```

When we run the cell, the output is the amount of time it took to run the code for 7 runs and 1,000 loops per run.

```
%timeit np.random.uniform(-90.000, 90.000, size=1500)
14.6 µs ± 526 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

The output is the amount of time it took to run this code, which is an average of 14.6 microseconds. The amount of time it takes to run this code depends on the processing speed and the RAM of your computer.

Now, let's run the `while` loop as a function. Copy the following code in a new cell and run the cell.

```
def latitudes(size):
    latitudes = []
    x = 0
    while x < (size):
```

```
        random_lat = random.randint(-90, 90) + random.random()
        latitudes.append(random_lat)
        x += 1
    return latitudes
# Call the function with 1500.
%timeit latitudes(1500)
```

The output is 1.45 milliseconds.

```
# Call the function with 1500.
%timeit latitudes(1500)
```

```
1.45 ms ± 5.39 us per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Using the `np.random.uniform(-90.000, 90.000, size=1500)` is 100 times faster than using the function, and our code is one line, whereas the function uses eight lines!

NOTE

For more information, see the **documentation on numpy.random.uniform() (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.uniform.html)** .

SKILL DRILL

Refactor the code for the while loop with the `%timeit` magic command and write a for loop that will generate the 1,500 latitudes.

Create Latitude and Longitude Combinations

Let's apply our new knowledge to this project. Create a new Jupyter Notebook file called `WeatherPy.ipynb`, import the Pandas, Matplotlib, and NumPy dependencies in the first cell, and run the cell.

```
# Import the dependencies.
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

In the next cell, we'll add the code that generates the latitudes and longitudes, but first, they need to be stored so that we can access them later. Since we are creating arrays of latitudes and longitudes, we'll declare each array as a variable.

In the next cell, add the following code that we used to generate the random latitudes. Also, we'll create a similar code snippet that will generate longitudes. To ensure enough latitudes and longitudes, we'll start with 1,500. In addition, we'll pack the latitudes (`lats`) and longitudes (`lngs`) as pairs by zipping them (`lat_lngs`) with the `zip()` function.

```
# Create a set of random latitude and longitude combinations.
lats = np.random.uniform(low=-90.000, high=90.000, size=1500)
lngs = np.random.uniform(low=-180.000, high=180.000, size=1500)
lat_lngs = zip(lats, lngs)
lat_lngs
```

When we run this cell, the output is a zip object in memory.

```
# Create a set of random latitude and longitude combinations.
lats = np.random.uniform(low=-90.000, high=90.000, size=1500)
lngs = np.random.uniform(low=-180.000, high=180.000, size=1500)
lat_lngs = zip(lats, lngs)
lat_lngs
```

```
<zip at 0x113dc66c8>
```

The zip object packs each pair of `lats` and `lngs` having the same index in their respective array into a tuple. If there are 1,500 latitudes and longitudes, there will be 1,500 tuples of paired latitudes and longitudes, where each

latitude and longitude in a tuple can be accessed by the index of 0 and 1, respectively.

Let's practice zipping a small number of latitudes and longitudes and then unpacking the zipped tuple to see how the packing and unpacking work.

In a new Jupyter Notebook file called `API_practice.ipynb`, add the following lists and pack them into the zipped tuple. Then, run the cell.

```
# Create a practice set of random latitude and longitude combinations.
lats = [25.12903645, 25.92017388, 26.62509167, -59.98969384, 37.30571269]
lngs = [-67.59741259, 11.09532135, 74.84233102, -76.89176677, -61.1337628
lat_lngs = zip(lats, lngs)
```

Next, let's unpack our `lat_lngs` zip object into a list. This way, we only need to create a set of random latitudes and longitudes once. In a new cell in the `WeatherPy.ipynb` file, add the following code and run the cell.

```
# Add the latitudes and longitudes to a list.
coordinates = list(lat_lngs)
```

NOTE

You can only unzip a zipped tuple once before it is removed from the computer's memory. Make sure you unzip the latitudes and longitudes into the coordinates list before moving on.

In a new cell, display the coordinate pairs with the following code.

```
# Use the print() function to display the latitude and longitude combinat
for coordinate in coordinates:
    print(coordinate[0], coordinate[1])
```

When we run the cell above, the output is ordered pairs of our `lats` and `lngs` coordinates.

```
25.12903645 -67.59741259
25.92017388 11.09532135
26.62509167 74.84233102
-59.98969384 -76.89176677
37.30571269 -61.13376282
```

Now that we have our ordered pairs of latitudes and longitudes in a list, we can iterate through the list of tuples and find the nearest city to those coordinates.