Modeling and Simulation in Python

Chapter 2

In [ ]:
```
# Configure Jupyter so figures appear in the notebook
%matplotlib inline

# Configure Jupyter to display the assigned value after an assignment
%config InteractiveShell.ast_node_interactivity='last_expr_or_assign'

# import functions from the modsim library
from modsim import *

# set the random number generator
np.random.seed(7)

# If this cell runs successfully, it produces no output.
```

Modeling a bikeshare system

We'll start with a `State` object that represents the number of bikes at each station.

When you display a `State` object, it lists the state variables and their values:

In [ ]:
```
bikeshare = State(olin=10, wellesley=2)
```

Out[ ]:

	values
olin	10
wellesley	2

We can access the state variables using dot notation.

In [ ]:
```
bikeshare.olin
```

Out[ ]: 10

In [ ]:
```
bikeshare.wellesley
```

Out[ ]: 2

Exercise: What happens if you spell the name of a state variable wrong? Edit the previous cell, change the spelling of `wellesley`, and run the cell again.

The error message uses the word "attribute", which is another name for what we are calling a state variable.

In [ ]:
```
try:
    bikeshare.wellesle
except:
    print('It returns an error!')
```

```
It returns an error!
```

RESPONSE: It returns an error!

Exercise: Add a third attribute called `babson` with initial value 0, and display the state of `bikeshare` again.

In [ ]:
```
bikeshare = State(olin=10, wellesley=2, babson = 0)
bikeshare
```

Out[ ]:

	values
olin	10
wellesley	2
babson	0

Updating

We can use the update operators `+=` and `-=` to change state variables.

In [ ]:
```
bikeshare.olin -= 1
```

If we display `bikeshare`, we should see the change.

In [ ]:
```
bikeshare
```

Out[ ]:

	values
olin	9
wellesley	2
babson	0

Of course, if we subtract a bike from `olin`, we should add it to `wellesley`.

In [ ]:
```
bikeshare.wellesley += 1
bikeshare
```

Out[ ]:

	values
olin	9
wellesley	3
babson	0

Functions

We can take the code we've written so far and encapsulate it in a function.

In [ ]:
```
def bike_to_wellesley():
    bikeshare.olin -= 1
    bikeshare.wellesley += 1
```

When you define a function, it doesn't run the statements inside the function, yet. When you call the function, it runs the statements inside.

In [ ]:
```
bike_to_wellesley()
bikeshare
```

Out[ ]:

	values
olin	8
wellesley	4
babson	0

One common error is to omit the parentheses, which has the effect of looking up the function, but not calling it.

In [ ]:
```
bike_to_wellesley
```

Out[ ]:
```
<function __main__.bike_to_wellesley()>
```

The output indicates that `bike_to_wellesley` is a function defined in a "namespace" called `__main__`, but you don't have to understand what that means.

Exercise: Define a function called `bike_to_olin` that moves a bike from Wellesley to Olin. Call the new function and display `bikeshare` to confirm that it works.

In [ ]:
```
def bike_to_olin():
    bikeshare.wellesley -= 1
    bikeshare.olin += 1
```

In [ ]:
```
bike_to_olin()
bikeshare
```

Out[ ]:

	values
olin	9
wellesley	3
babson	0

Conditionals

`modsim.py` provides `flip`, which takes a probability and returns either `True` or `False`, which are special values defined by Python.

The Python function `help` looks up a function and displays its documentation.

In [ ]:
```
help(flip)
```

```
Help on function flip in module modsim.modsim:

flip(p=0.5)
    Flips a coin with the given probability.

    p: float 0-1

    returns: boolean (True or False)
```

In the following example, the probability is 0.7 or 70%. If you run this cell several times, you should get `True` about 70% of the time and `False` about 30%.

In [ ]:
```
flip(0.7)
```

Out[ ]:
```
True
```

In the following example, we use `flip` as part of an if statement. If the result from `flip` is `True`, we print `heads`; otherwise we do nothing.

In [ ]:
```
if flip(0.7):
    print('heads')
```

With an else clause, we can print heads or tails depending on whether `flip` returns `True` or `False`.

In [ ]:
```
if flip(0.7):
    print('heads')
else:
    print('tails')
```

```
heads
```

Step

Now let's get back to the bikeshare state. Again let's start with a new `State` object.

In [ ]:
```
bikeshare = State(olin=10, wellesley=2)
```

Out[ ]:

	values
olin	10
wellesley	2

Suppose that in any given minute, there is a 50% chance that a student picks up a bike at Olin and rides to Wellesley. We can simulate that like this.

In [ ]:
```
if flip(0.5):
    bike_to_wellesley()
    print('Moving a bike to Wellesley')

bikeshare
```

Out[ ]:

	values
olin	10
wellesley	2

And maybe at the same time, there is also a 40% chance that a student at Wellesley rides to Olin.

In [ ]:
```
if flip(0.4):
    bike_to_olin()
    print('Moving a bike to Olin')

bikeshare
```

Out[ ]:

	values
olin	10
wellesley	2

We can wrap that code in a function called `step` that simulates one time step. In any given minute, a student might ride from Olin to Wellesley, from Wellesley to Olin, or both, or neither, depending on the results of `flip`.

In [ ]:
```
def step():
    if flip(0.5):
```

```
        bike_to_wellesley()
        print('Moving a bike to Wellesley')

    if flip(0.4):
        bike_to_olin()
        print('Moving a bike to Olin')
```

Since this function takes no parameters, we call it like this:

In [ ]:
```
step()
bikeshare
```

Out[ ]:

	values
olin	10
wellesley	2

Parameters

As defined in the previous section, `step` is not as useful as it could be, because the probabilities `0.5` and `0.4` are "hard coded".

It would be better to generalize this function so it takes the probabilities `p1` and `p2` as parameters:

In [ ]:
```
def step(p1, p2):
    if flip(p1):
        bike_to_wellesley()
        print('Moving a bike to Wellesley')

    if flip(p2):
        bike_to_olin()
        print('Moving a bike to Olin')
```

Now we can call it like this:

In [ ]:
```
step(0.5, 0.4)
bikeshare
```

```
Moving a bike to Wellesley
Moving a bike to Olin
```

Out[ ]:

	values
olin	10
wellesley	2

Exercise: At the beginning of `step`, add a print statement that displays the values of `p1` and `p2`. Call it again with values `0.3`, and `0.2`, and confirm that the values of the parameters are what you expect.

In [ ]:
```
def step(p1, p2):
    print(f'p1 = {p1}, p2 = {p2}')

    if flip(p1):
        bike_to_wellesley()
        print('Moving a bike to Wellesley')

    if flip(p2):
        bike_to_olin()
        print('Moving a bike to Olin')

step(0.3, 0.2)
```

```
p1 = 0.3, p2 = 0.2
```

For loop

Before we go on, I'll redefine `step` without the print statements.

In [ ]:
```
def step(p1, p2):
    if flip(p1):
        bike_to_wellesley()

    if flip(p2):
        bike_to_olin()
```

And let's start again with a new `State` object:

In [ ]:
```
bikeshare = State(olin=10, wellesley=2)
```

Out[ ]:

	values
olin	10
wellesley	2

We can use a `for` loop to move 4 bikes from Olin to Wellesley.

In [ ]:
```
for i in range(4):
    bike_to_wellesley()

bikeshare
```

Out[ ]:

	values
olin	6
wellesley	6

Or we can simulate 4 random time steps.

In [ ]:
```
for i in range(4):
    step(0.3, 0.2)

bikeshare
```

Out[ ]:

	values
olin	5
wellesley	7

If each step corresponds to a minute, we can simulate an entire hour like this.

In [ ]:
```
for i in range(60):
    step(0.3, 0.2)

bikeshare
```

Out[ ]:

	values
olin	4
wellesley	8

After 60 minutes, you might see that the number of bike at Olin is negative. We'll fix that problem in the next notebook.

But first, we want to plot the results.

TimeSeries

`modsim.py` provides an object called a `TimeSeries` that can contain a sequence of values changing over time.

We can create a new, empty `TimeSeries` like this:

In [ ]:
```
results = TimeSeries()
```

Out[ ]:

values

And we can add a value to the `TimeSeries` like this:

In [ ]:
```
results[0] = bikeshare.olin
results
```

Out[ ]:

	values
0	4

The `0` in brackets is an `index` that indicates that this value is associated with time step 0.

Now we'll use a for loop to save the results of the simulation. I'll start one more time with a new `State` object.

In [ ]:
```
bikeshare = State(olin=10, wellesley=2)
```

Out[ ]:

	values
olin	10
wellesley	2

Here's a for loop that runs 10 steps and stores the results.

In [ ]:
```
for i in range(10):
    step(0.3, 0.2)
    results[i] = bikeshare.olin
```

Now we can display the results.

In [ ]:
```
results
```

Out[ ]:

	values
0	10
1	10
2	9
3	9
4	10
5	10
6	10
7	11
8	11
9	11

A `TimeSeries` is a specialized version of a Pandas `Series`, so we can use any of the functions provided by `Series`, including several that compute summary statistics:

In [ ]:
```
results.mean()
```

Out[ ]:
```
10.1
```

In [ ]:
```
results.describe()
```

Out[ ]:
```
count    10.000000
mean     10.100000
std       0.737865
min       9.000000
25%      10.000000
50%      10.000000
75%      10.750000
max      11.000000
dtype: float64
```

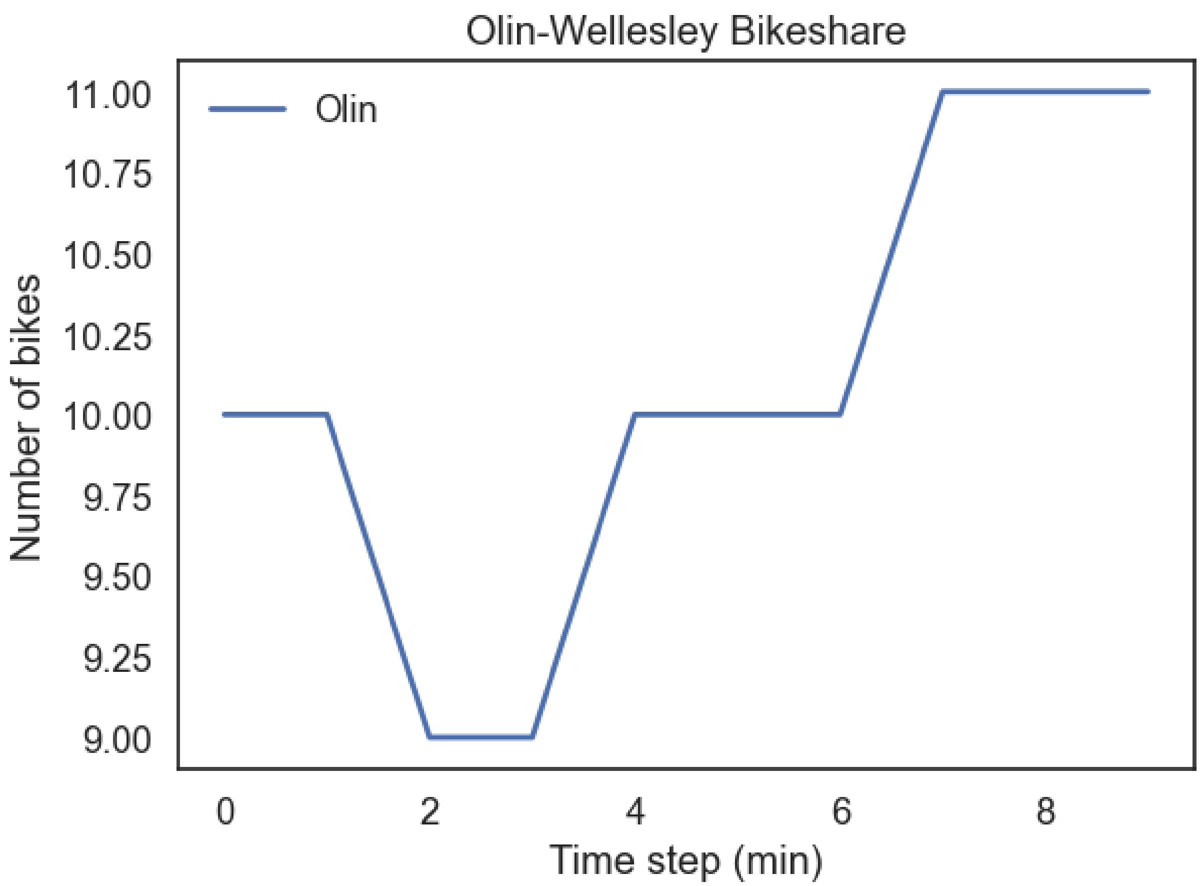
You can read the documentation of `Series` here.

Plotting

We can also plot the results like this.

In [ ]:
```
plot(results, label='Olin')

decorate(title='Olin-Wellesley Bikeshare',
         xlabel='Time step (min)',
         ylabel='Number of bikes')

savefig('figs/chap02-fig01.pdf')
```

```
Saving figure to file figs/chap02-fig01.pdf
```



`decorate`, which is defined in the `modsim` library, adds a title and labels the axes.

In [ ]:
```
help(decorate)
```

```
Help on function decorate in module modsim.modsim:

decorate(**options)
    Decorate the current axes.

    Call decorate with keyword arguments like

    decorate(title='Title',
             xlabel='x',
             ylabel='y')

    The keyword arguments can be any of the axis properties

    https://matplotlib.org/api/axes_api.html

    In addition, you can use `legend=False` to suppress the legend.

    And you can use `loc` to indicate the location of the legend
    (the default value is 'best')
```

`savefig()` saves a figure in a file.

In [ ]:
```
help(savefig)
```

```
Help on function savefig in module modsim.modsim:

savefig(filename, **options)
    Save the current figure.

    Keyword arguments are passed along to plt.savefig

    https://matplotlib.org/api/_as_gen/matplotlib.pyplot.savefig.html

    filename: string
```

The suffix of the filename indicates the format you want. This example saves the current figure in a PDF file.

Exercise: Wrap the code from this section in a function named `run_simulation` that takes three parameters, named `p1`, `p2`, and `num_steps`.

It should:

1. Create a `TimeSeries` object to hold the results.
2. Use a for loop to run `step` the number of times specified by `num_steps`, passing along the specified values of `p1` and `p2`.
3. After each step, it should save the number of bikes at Olin in the `TimeSeries`.
4. After the for loop, it should plot the results and
5. Decorate the axes.

To test your function:

1. Create a `State` object with the initial state of the system.
2. Call `run_simulation` with appropriate parameters.
3. Save the resulting figure.

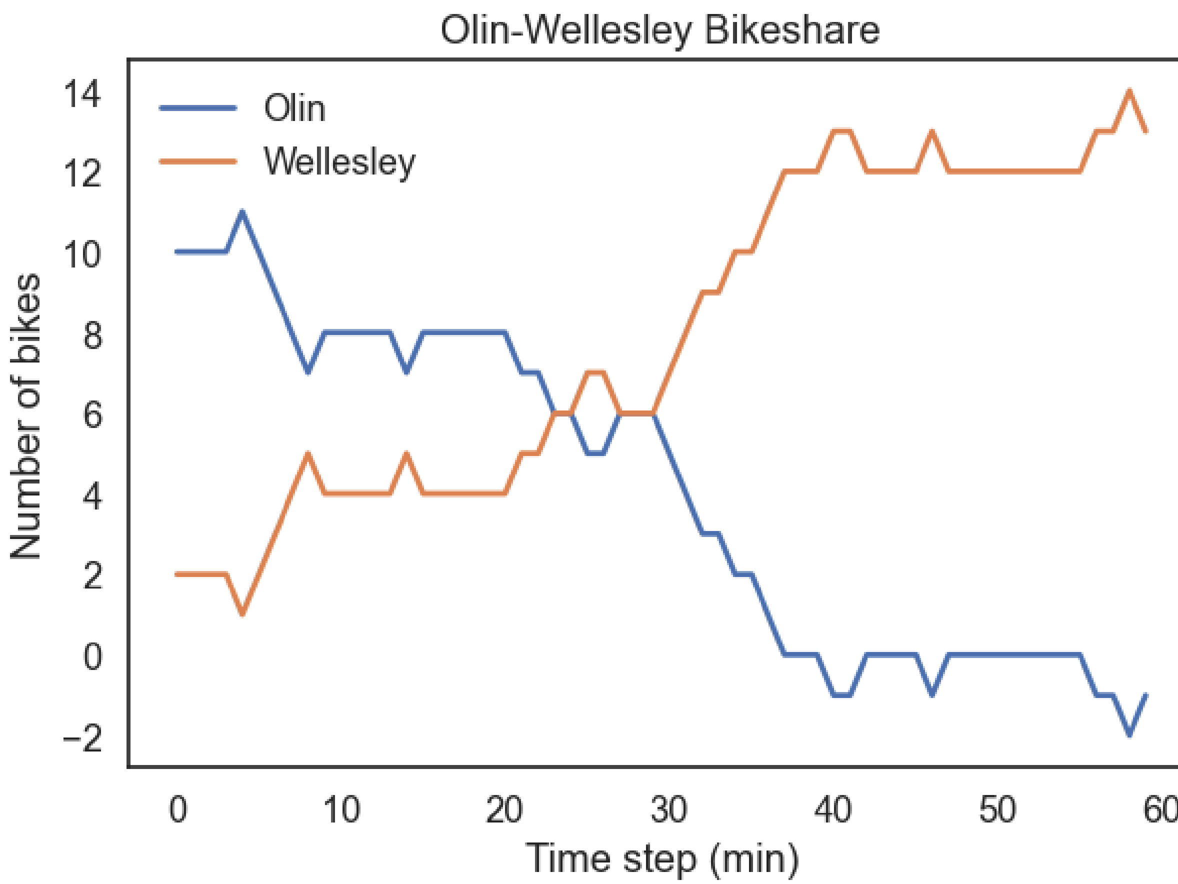
Optional:

1. Extend your solution so it creates two `TimeSeries` objects, keeps track of the number of bikes at Olin *and* at Wellesley, and plots both series at the end.

In [ ]:
```
def run_simulation(state, p1, p2, num_steps):
    # Initialize results object
    results_olin = TimeSeries()
    results_wellesley = TimeSeries()

    # Run the simulation
    for i in range(num_steps):
        step(p1, p2)
        results_olin[i] = state.olin
        results_wellesley[i] = state.wellesley

    # Plot the results
    plot(results_olin, label='Olin')
```

```
    decorate(title='Olin-Wellesley Bikeshare',
             xlabel='Time step (min)',
             ylabel='Number of bikes')

    plot(results_wellesley, label='Wellesley')
    decorate(title='Olin-Wellesley Bikeshare',
             xlabel='Time step (min)',
             ylabel='Number of bikes')
```

In [ ]:
```
bikeshare = State(olin=10, wellesley=2)
run_simulation(bikeshare, 0.3, 0.2, 60)
```



Opening the hood

The functions in `modsim.py` are built on top of several widely-used Python libraries, especially NumPy, SciPy, and Pandas. These libraries are powerful but can be hard to use. The intent of `modsim.py` is to give you the power of these libraries while making it easy to get started.

In the future, you might want to use these libraries directly, rather than using `modsim.py`. So we will pause occasionally to open the hood and let you see how `modsim.py` works.

You don't need to know anything in these sections, so if you are already feeling overwhelmed, you might want to skip them. But if you are curious, read on.

Pandas

This chapter introduces two objects, `State` and `TimeSeries`. Both are based on the `Series` object defined by Pandas, which is a library primarily used for data science.

You can read the documentation of the `Series` object here

The primary differences between `TimeSeries` and `Series` are:

1. I made it easier to create a new, empty `Series` while avoiding a confusing inconsistency.

2. I provide a function so the `Series` looks good when displayed in Jupyter.

3. I provide a function called `set` that we'll use later.

`State` has all of those capabilities; in addition, it provides an easier way to initialize state variables, and it provides functions called `T` and `dt`, which will help us avoid a confusing error later.

Pyplot

The `plot` function in `modsim.py` is based on the `plot` function in Pyplot, which is part of Matplotlib. You can read the documentation of `plot` here.

`decorate` provides a convenient way to call the `pyplot` functions `title`, `xlabel`, and `ylabel`, and `legend`. It also avoids an annoying warning message if you try to make a legend when you don't have any labelled lines.

In [ ]:

```
help(decorate)
```

```
Help on function decorate in module modsim.modsim:

decorate(**options)
    Decorate the current axes.

    Call decorate with keyword arguments like

    decorate(title='Title',
             xlabel='x',
             ylabel='y')

    The keyword arguments can be any of the axis properties

    https://matplotlib.org/api/axes_api.html

    In addition, you can use `legend=False` to suppress the legend.

    And you can use `loc` to indicate the location of the legend
    (the default value is 'best')
```

NumPy

The `flip` function in `modsim.py` uses NumPy's `random` function to generate a random number between 0 and 1.

You can get the source code for `flip` by running the following cell.

In [ ]:
```
source_code(flip)
```

```
def flip(p=0.5):
    """Flips a coin with the given probability.

    p: float 0-1

    returns: boolean (True or False)
    """
    return np.random.random() < p
```

In [ ]:
```

```