Modeling and Simulation in Python

Chapter 1

Jupyter

Welcome to *Modeling and Simulation*, welcome to Python, and welcome to Jupyter.

This is a Jupyter notebook, which is a development environment where you can write and run Python code. Each notebook is divided into cells. Each cell contains either text (like this cell) or Python code.

Selecting and running cells

To select a cell, click in the left margin next to the cell. You should see a blue frame surrounding the selected cell.

To edit a code cell, click inside the cell. You should see a green frame around the selected cell, and you should see a cursor inside the cell.

To edit a text cell, double-click inside the cell. Again, you should see a green frame around the selected cell, and you should see a cursor inside the cell.

To run a cell, hold down SHIFT and press ENTER.

- If you run a text cell, Jupyter formats the text and displays the result.
- If you run a code cell, Jupyter runs the Python code in the cell and displays the result, if any.

To try it out, edit this cell, change some of the text, and then press SHIFT-ENTER to format it.

Adding and removing cells

You can add and remove cells from a notebook using the buttons in the toolbar and the items in the menu, both of which you should see at the top of this notebook.

Try the following exercises:

1. From the Insert menu select "Insert cell below" to add a cell below this one. By default, you get a code cell, as you can see in the pulldown menu that says "Code".
2. In the new cell, add a print statement like `print('Hello')`, and run it.
3. Add another cell, select the new cell, and then click on the pulldown menu that says "Code" and select "Markdown". This makes the new cell a text cell.
4. In the new cell, type some text, and then run it.
5. Use the arrow buttons in the toolbar to move cells up and down.
6. Use the cut, copy, and paste buttons to delete, add, and move cells.
7. As you make changes, Jupyter saves your notebook automatically, but if you want to make sure, you can press the save button, which looks like a floppy disk from the 1990s.
8. Finally, when you are done with a notebook, select "Close and Halt" from the File menu.

Using the notebooks

The notebooks for each chapter contain the code from the chapter along with additional examples, explanatory text, and exercises. I recommend you

1. Read the chapter first to understand the concepts and vocabulary,
2. Run the notebook to review what you learned and see it in action, and then
3. Attempt the exercises.

If you try to work through the notebooks without reading the book, you're gonna have a bad time. The notebooks contain some explanatory text, but it is probably not enough to make sense if you have not read the book. If you are working through a notebook and you get stuck, you might want to re-read (or read!) the corresponding section of the book.

Installing modules

These notebooks use standard Python modules like NumPy and SciPy. I assume you already have them installed in your environment.

They also use two less common modules: Pint, which provides units, and modsim, which contains code I wrote specifically for this book.

The following cells check whether you have these modules already and tries to install them if you don't.

In [ ]:
```
# try:
#     import pint
# except ImportError:
```

```
#     !pip install pint
#     import pint
```

In [ ]:
```
# try:
#     from modsim import *
# except ImportError:
#     %pip install modsimpy
#     from modsim import *
```

In [ ]:
```
import pint
from modsim import *
```

The first time you run this on a new installation of Python, it might produce a warning message in pink. That's probably ok, but if you get a message that says `modsim.py depends on Python 3.7 features`, that means you have an older version of Python, and some features in `modsim.py` won't work correctly.

If you need a newer version of Python, I recommend installing Anaconda. You'll find more information in the preface of the book.

You can find out what version of Python and Jupyter you have by running the following cells.

In [ ]:
```
!python --version
```

```
Python 3.11.7
```

In [ ]:
```
!jupyter-notebook --version
```

```
'jupyter-notebook' is not recognized as an internal or external command,
operable program or batch file.
```

Configuring Jupyter

The following cell:

1. Uses a Jupyter "magic command" to specify whether figures should appear in the notebook, or pop up in a new window.

2. Configures Jupyter to display some values that would otherwise be invisible.

Select the following cell and press SHIFT-ENTER to run it.

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'
```

The penny myth

The following cells contain code from the beginning of Chapter 1.

`modsim` defines `UNITS`, which contains variables representing pretty much every unit you've ever heard of. It uses Pint, which is a Python library that provides tools for computing with units.

The following lines create new variables named `meter` and `second`.

In [ ]:
```
meter = UNITS.meter
```

Out[ ]: meter

In [ ]:
```
second = UNITS.second
```

Out[ ]: second

To find out what other units are defined, type `UNITS.` (including the period) in the next cell and then press TAB. You should see a pop-up menu with a list of units.

In [ ]:
```
units = [x for x in UNITS]
units[200:220]
```

Out[ ]:
```
['astronomical_unit',
 'at',
 'atm',
 'atm_l',
 'atmosphere',
 'atmosphere_liter',
 'atomic_mass_constant',
 'atomic_unit_of_action',
 'atomic_unit_of_current',
 'atomic_unit_of_electric_field',
 'atomic_unit_of_energy',
 'atomic_unit_of_force',
 'atomic_unit_of_intensity',
 'atomic_unit_of_length',
 'atomic_unit_of_mass',
 'atomic_unit_of_temperature',
 'atomic_unit_of_time',
 'au',
 'avdp_dram',
 'avdp_ounce']
```

Create a variable named `a` and give it the value of acceleration due to gravity.

In [ ]:
```
a = 9.8 * meter / second**2
```

Out[ ]: 9.8 meter/second^2

Create `t` and give it the value 4 seconds.

In [ ]:
```
t = 4 * second
```

Out[ ]: 4 second

Compute the distance a penny would fall after `t` seconds with constant acceleration `a`. Notice that the units of the result are correct.

In [ ]:
```
a * t**2 / 2
```

Out[ ]: 78.4 meter

Exercise: Compute the velocity of the penny after `t` seconds. Check that the units of the result are correct.

In [ ]:
```
a * t
```

Out[ ]: 39.2 meter/second

Exercise: Why would it be nonsensical to add `a` and `t`? What happens if you try?

RESPONSE: They are different units!

In [ ]:
```
try:
    a + t
except:
    print('a and t are incompatible')
```

```
a and t are incompatible
```

The error messages you get from Python are big and scary, but if you read them carefully, they contain a lot of useful information.

1. Start from the bottom and read up.
2. The last line usually tells you what type of error happened, and sometimes additional information.
3. The previous lines are a "traceback" of what was happening when the error occurred. The first section of the traceback shows the code you wrote. The following sections are often from Python libraries.

In this example, you should get a `DimensionalityError`, which is defined by Pint to indicate that you have violated a rules of dimensional analysis: you cannot add quantities with different dimensions.

Before you go on, you might want to delete the erroneous code so the notebook can run without errors.

Falling pennies

Now let's solve the falling penny problem.

Set `h` to the height of the Empire State Building:

In [ ]:
```
h = 381 * meter
```

Out[ ]: 381 meter

Compute the time it would take a penny to fall, assuming constant acceleration.

$at^2/2 = h$

$t = \sqrt{2h/a}$

In [ ]:
```
t = sqrt(2 * h / a)
```

Out[ ]: 8.817885349720552 second

Given `t`, we can compute the velocity of the penny when it lands.

$v = at$

In [ ]:
```
v = a * t
```

Out[ ]: 86.41527642726142 meter/second

We can convert from one set of units to another like this:

In [ ]:
```
mile = UNITS.mile
hour = UNITS.hour
```

Out[ ]: hour

In [ ]:
```
v.to(mile/hour)
```

Out[ ]: 193.30546802805432 mile/hour

Exercise: Suppose you bring a 10 foot pole to the top of the Empire State Building and use it to drop the penny from `h` plus 10 feet.

Define a variable named `foot` that contains the unit `foot` provided by `UNITS`. Define a variable named `pole_height` and give it the value 10 feet.

What happens if you add `h`, which is in units of meters, to `pole_height`, which is in units of feet? What happens if you write the addition the other way around?

In [ ]:
```
foot = UNITS.foot
pole_height = 10 * foot
```

Out[ ]: 10 foot

In [ ]:
```
h + pole_height
```

Out[ ]: 384.048 meter

In [ ]:
```
pole_height + h
```

Out[ ]: 1260.0 foot

RESPONSE: If two variables measuring the same aspect (e.g. height) are added, then the result takes the unit of the first variable in the operation. So, `h + pole_height` is expressed in meters, because `h` is in meters. By contrast, `pole_height + h` is expressed in feet, because `pole_height` is in feet.

Exercise: In reality, air resistance limits the velocity of the penny. At about 18 m/s, the force of air resistance equals the force of gravity and the penny stops accelerating.

As a simplification, let's assume that the acceleration of the penny is `a` until the penny reaches 18 m/s, and then 0 afterwards. What is the total time for the penny to fall 381 m?

You can break this question into three parts:

1. How long until the penny reaches 18 m/s with constant acceleration `a`?
2. How far would the penny fall during that time?
3. How long to fall the remaining distance with constant velocity 18 m/s?

Suggestion: Assign each intermediate result to a variable with a meaningful name. And assign units to all quantities!

In [ ]:
```
# How long until the penny reaches 18 m/s with constant acceleration `a`?
# v = a * t
# t = v / a

v = 18 * meter / second
time_to_terminal_velocity = v / a
```

Out[ ]: 1.8367346938775508 second

In [ ]:
```
# How far would the penny fall during that time?
# t = sqrt(2 * h / a)
# h = a * t**2 / 2

distance_during_ttv = a * time_to_terminal_velocity**2 / 2
```

Out[ ]: 16.530612244897956 meter

In [ ]:
```
# How long to fall the remaining distance with constant velocity 18 m/s?

remaining_distance = h + pole_height - distance_during_ttv
time_to_ground = remaining_distance / v
```

Out[ ]: 20.417632653061226 second

Restart and run all

When you change the contents of a cell, you have to run it again for those changes to have an effect. If you forget to do that, the results can be confusing, because the code you are looking at is not the code you ran.

If you ever lose track of which cells have run, and in what order, you should go to the Kernel menu and select "Restart & Run All". Restarting the kernel means that all of your variables get deleted, and running all the cells means all of your code will run again, in the right order.

Exercise: Select "Restart & Run All" now and confirm that it does what you want.

In [ ]:
```
print('It works!')
```

```
It works!
```