getting-started

October 23, 2019

This notebook is adapted from course material from CBE20255 by Jeffrey Kantor (jeff at nd.edu); see original content on Github. The text is released under the CC-BY-NC-ND-4.0 license, and code is released under the MIT license.

1 Jupyter Notebooks

Jupyter Notebooks:

- are interactive documents, like this one, that include formatted text and code
- are simple JSON documents with a suffix `.ipynb`
- can be uploaded, downloaded, and shared like any other digital document
- are composed of individual cells containing either text (Markdown format), code, the output of a calculation, or raw "NBConvert" content

The code cells can be written in different programming languages such as Python, R, and Julia.

1.1 Markdown cells

Markdown is a popular markup language that is a superset of HTML. Its complete specification can be found here.

With Markdown, you can format the text portion of your notebook however you like.

For example, you can make headings...

```
#### This is a Fourth-Level Heading
```

This is a Fourth-Level Heading

Or, you can make a list...

```
- first item
- second item
- third item
```

- first item
- second item
- third item

Or, you can use LaTeX for math equations...

```
$E = m c^2$
```

$E = mc^2$

1.2 Code cells

These are cells containing Python (or R or an other language depending on what your Jupyter installation supports).
You decide when you create the notebook.

For example, the following cell demonstrates some basic Python types, arithmetic, and printing.

[1]:
```
a = 12
b = 2
print('a + b = {}'.format(a + b))
print('type(a+b): {}'.format(type(a+b)))

print('\na**b = {}'.format(a**b))
print('type(a**b): {}'.format(type(a**b)))

print('\na/b = {}'.format(a/b))
print('type(a/b): {}'.format(type(a/b)))
```

```
a + b = 14
type(a+b): <class 'int'>

a**b = 144
type(a**b): <class 'int'>

a/b = 6.0
type(a/b): <class 'float'>
```

1.3 Raw NBConvert cells

A raw cell is defined as content that should be included unmodified when the notebook is converted with the commandline tool `nbconver` or via the `File -> Download As` dialog.

For example, an NBConvert cell could include raw LaTeX for use when creating a pdf or restructured text for use in Sphinx documentation.

In this notebook, there are NBConvert cells with the LaTeX command `\newpage` for adding page breaks when the file is converted to a pdf with `pdfLatex`.

2 Modules

One of the best features of Python is the ability to extend it's functionality by importing special purpose libraries of functions, or modules. Three different ways to define a module in Python:

- A module can be written in Python itself.
- A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.
- A built-in module is intrinsically contained in the interpreter, like the itertools module.
- A module's contents are accessed the same way in all three cases: with the import statement.

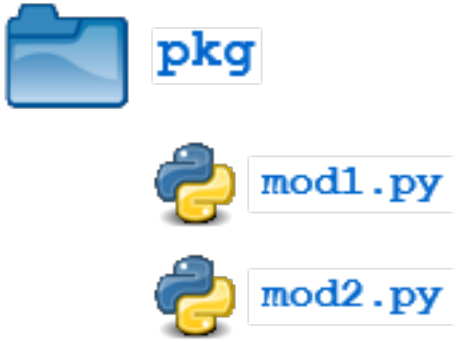
3 Packages

- A way to organize large modules of methods, functions, classes, etc, into separate namespaces.
- Hierarchical dot namespace notation.
- Also helps by allowing you to import only specific portions of your code, saving memory.

3.1 Package heirarchy = file system heirarchy

Packages use the operating system's inherent hierarchical file structure.

```
import pkg.mod1, pkg.mod2
pkg.mod1.method_in_mod1()
pkg.mod2.method_in_mod2()
```



4 Some Basic Python for Science and Engineering

4.1 Scientific computing with numpy

The numpy library is the fundamental package for scientific computing with Python. It contains, among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

This next cell shows how to import `numpy` with the prefix `np`, then use it to call a common function:

[2]:
```
import numpy as np
np.sin(np.pi/2)
```

[2]:
```
1.0
```

4.2 Lists

Lists are a versatile way of organizing your data in Python.

[3]:
```
xList = [1, 2, 3, 4]
print(xList)
```

```
[1, 2, 3, 4]
```

Concatentation is the operation of joining one list to another.

[4]:
```
# Concatenation
x = [1, 2, 3, 4];
y = [5, 6, 7, 8];

x + y
```

[4]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

A common operation is to apply a binary operation to elements of a single list. For an example such as arithmetic addition, this can be done using the `sum` function from `numpy`.

[5]:
```
# Two ways to sum a list of numbers
print(np.sum(x))
```

```
10
```

A **for loop** is a means for iterating over the elements of a list. The colon marks the start of code that will be executed for each element of a list.

Indentation is very important in Python.

In the cell below, everything in the indented block will be executed on each iteration of the for loop.

[6]:
```
for x in xList:
    print('x: {}\tsin(x): {}'.format(x, np.sin(np.pi/x)))
```

```
x: 1    sin(x): 1.2246467991473532e-16
x: 2    sin(x): 1.0
x: 3    sin(x): 0.8660254037844386
x: 4    sin(x): 0.7071067811865475
```

List comprehension is a very useful tool for creating a new list using data from an existing list. For example, suppose you want a list consisting random numbers. The `random.random()` function below returns a random floating point number in the range $[0.0, 1.0)$.

[7]:
```
from random import random

[i + random() for i in range(0,10)]
```

[7]:
```
[0.9655671267273181,
 1.7768231524796403,
 2.111193336375665,
 3.277954787209489,
 4.744107677608763,
 5.62074341883594,
 6.633566843920021,
 7.514551643135531,
 8.469561311177692,
 9.478905635245377]
```

4.3 Dictionaries

Dictionaries are useful for storing and retrieving data as **key-value pairs**. For example, here is a short dictionary of molar masses. The keys are molecular formulas, and the values are the corresponding molar masses.

[8]:
```
mw = {'CH4': 16.04, 'H2O': 18.02, 'O2':32.00, 'CO2': 44.01}
print(mw)
```

```
{'CH4': 16.04, 'H2O': 18.02, 'O2': 32.0, 'CO2': 44.01}
```

We can add a value to an existing dictionary by simply specifying the variable with a new key and equating it to a value.

[9]:
```
mw['C8H18'] = 114.23
print(mw)
```

```
{'CH4': 16.04, 'H2O': 18.02, 'O2': 32.0, 'CO2': 44.01, 'C8H18': 114.23}
```

We can retrieve a value from a dictionary.

[10]:
```
mw['CH4']
```

[10]:
```
16.04
```

A for loop is a useful means of interating over all key-value pairs of a dictionary.

[11]:
```
for species in mw.keys():
    print('The molar mass of {:<s} is {:<7.2f}'.format(species, mw[species]))
```

```
The molar mass of CH4 is 16.04
The molar mass of H2O is 18.02
The molar mass of O2 is 32.00
The molar mass of CO2 is 44.01
The molar mass of C8H18 is 114.23
```

Dictionaries can be sorted by key or by value.

[12]:
```
for species in sorted(mw):
    print(" {:<8s}  {:>7.2f}".format(species, mw[species]))
```

```
 C8H18      114.23
 CH4         16.04
 CO2         44.01
 H2O         18.02
 O2          32.00
```

[13]:
```
for species in sorted(mw, key = mw.get):
    print(" {:<8s}  {:>7.2f}".format(species, mw[species]))
```

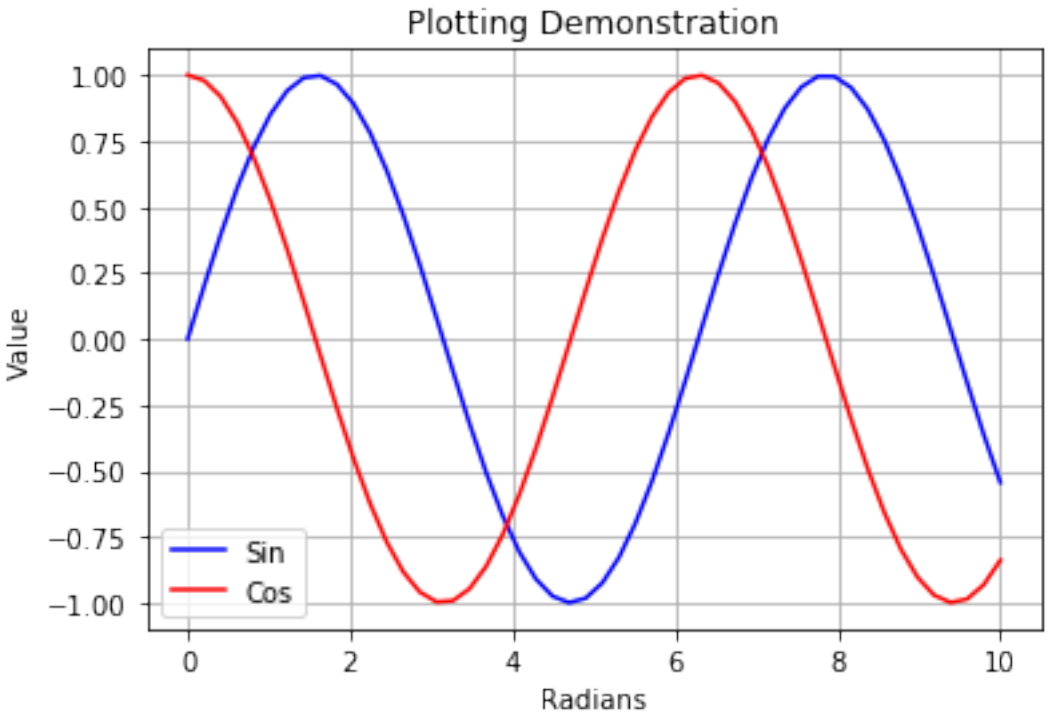
```
 CH4         16.04
 H2O         18.02
 O2          32.00
 CO2         44.01
 C8H18      114.23
```

4.4 Plotting with matplotlib

Importing the `matplotlib.pyplot` library gives Jupyter notebooks plotting functionality very similar to Matlab's. Here are some examples using functions from the library.

[14]:
```
import matplotlib.pyplot as plt
import numpy as np # note: this isn't necessary to do again in the same Python session
# sets matplotlib backend to 'inline'; if omitted, you need a plt.show() command
%matplotlib inline

x = np.linspace(0,10)
y = np.sin(x)
z = np.cos(x)
plt.plot(x,y,'b', x,z,'r')
plt.xlabel('Radians');
plt.ylabel('Value');
plt.title('Plotting Demonstration')
plt.legend(['Sin','Cos'])
plt.grid(True)
```


Plotting Demonstration
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Value
Sin
Cos
0
2
4
6
8
10
Radians

[15]:
```
plt.subplot(2,1,1) # 2 rows, 1 column, 1st plot
plt.plot(x,y)
plt.title('Sin(x)')
plt.subplot(2,1,2) # 2 rows, 1 column, 2nd plot
plt.plot(x,z)
plt.title('Cos(x)');
```