(U) The **logging** module has several levels, including `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`, which are just integer constants defined in the module. Custom levels can bee added with the `addLevelName` method, but the default levels should usually be sufficient. Each of the module-level functions `warning`, `critical`, and `info` is shorthand for a collection of functionality, which can be accessed through separate methods if fine-grained control is needed.

- Create a log message with severity at the level indicated (the method `log(level, message)` could also be used).
- Send that message to the `root` **logger**
- If the `root` logger is configured to accept messages of that level (or lower), send the message to the default **handler**, which formats the message and prints it to the console.

(U) This explains why the call to `logging.info` did not print to the screen: its severity is not high enough. By default, the root logger is set to only handle messages at the `WARNING` level or higher. The level of the root logger can be configured at the module level, but must be done *before* any messages are sent. Otherwise, the level must be set directly on the logger.

```
[(level, getattr(logging,level)) for level in ['DEBUG','INFO', 'WARNING', 'ERROR','CRITICAL']]
```

```
logging.root.getEffectiveLevel()
```

```
logging.root.getEffectiveLevel() == logging.WARNING # True
```

```
logging.root.setLevel(logging.INFO)
```

```
logging.root.getEffectiveLevel() == logging.INFO # True
```

```
logging.info("Now this should get logged.")
```

```
logging.log(21,"This will also get logged at a numbered custom level")
```

```
logging.basicConfig(level=logging.DEBUG)
```

```
logging.root.getEffectiveLevel() == logging.DEBUG # False
```

```
exit()
```

```
# New Session
```

```
import logging
```

```
logging.basicConfig(level=logging.INFO)
```

```
logging.info("This is some information.")
```

```
logging.root.getEffectiveLevel() == logging.INFO # True
```

(U) From all this, a strategy emerges around using **logging** to improve your debugging.

- Instead of `print` statements, add calls to `logging.debug` to your code.
- At the top of your script, use `logging.basicConfig(level=logging.DEBUG)` during development; switch to `level=logging.INFO` or `level=logging.WARNING` for production.
- Optionally, make a command-line option for your script that enables debugging output (e.g. `my_script.py --verbose`).

(U) So far, we have dealt with only the defaults: the logger `logging.root` , along with its associated `Handler` and `Formatter` . A program can create multiple loggers, handlers, and formatters and connect them all together in different ways. Aside from the `StreamHandler` already encountered, the `FileHandler` is also very common; it writes messages to a file named in its constructor. For ease of exposition, we use only StreamHandlers in our examples. Each handler has a `setLevel` method; a handler acts on a message only only if both the `logger` and the `handler` have levels set at or below the severity level of the message. Unless `logging.basicConfig()` has been called, handlers and formatters must be explicitly defined.

```
logging.basicConfig()

warnlog = logging.getLogger('warnings')

warnlog.setLevel(logging.WARN)

infolog = logging.getLogger('info')

infolog.setLevel(logging.INFO)

infolog.info('An informational message')

info_handler = logging.StreamHandler()

infolog.addHandler(info_handler)

qm_formatter = logging.Formatter('%(name)s???%(message)s???')

info_handler.setFormatter(qm_formatter)

info_handler.setLevel(logging.ERROR)

warnlog.warning('There it goes')

warnlog.info('Not there anymore')

infolog.info("It's coming back")

infolog.error("Oops, it didn't make it")
```

(U) In this example, `infolog` has two handlers: the default handler created by `basicConfig` and the explicitly set `info_handler` with its distinctive `???` -inspired formatter. This second handler only logs messages with severity equal to or higher than `logging.ERROR`, even though the `infolog` passes it messages with severity level as low as `logging.INFO`, as can be seen by the fact that the default handler prints these messages.

(U) Advanced Usage

(U) A variety of handlers have been written for different purposes. The `RotatingFileHandler` from the `logging.handlers` submodule automatically starts new log files when they reach a size, and keeps only a specified number of backups. The `TimedRotatingFileHandler` is similar, but time-based instead of size-based. Other handlers write messages to sockets, email, and over HTTP, TCP, or UDP.

(U) Formatters use old-style string formatting with `%`, and have access to a dictionary which contains several interesting properties, including the name of the logger, the level of severity, the current time, and other information about the environment surrounding the logging message. Custom keys can be passed to the formatter with a dictionary passed via the `extra` keyword in `logging.log` or related shortcut methods, e.g. `logging.warning`.

(U) If the configured levels are too restrictive, custom levels can be added. They must be assigned a number, which determines when messages at that level will be handled. No shortcut method is added for custom levels, so calls must be made to the `log` method. Of course, it's easy to add such a shortcut to the Logger class.

```
INFOWARN = 25
```

```
logging.addLevelName(INFOWARN, 'INFOWARN')
```

```
def infowarn(self, message):
    self.log(INFOWARN, message)
```

```
logging.Logger.infowarn = infowarn
```

```
logger = logging.getLogger('info.warn')
```

```
logger.infowarn("Halfway between info and warning.")
```

Module: Math and More

(b)(3)-P.L. 86-36

0

Updated about 2 years ago by [redacted] in COMP 3321

3 171 44

fcs6 python

(U) Module: Math and More

Recommendations

Math and More

The Math Module is extremely powerful... for doing math-y things. This page is mostly to demonstrate some of the neat things Python can do that are math related. We'll start with the math module.

You can take this a step further and investigate `cmath` for complex operations. Most of what is in `math` is also in `cmath`.

math.py

```
import math
print(dir(math))
```

Constants of note

ππ (Greek letter pi): Ratio of circle's circumference to its diameter.

```
math.pi
```

Euler's number, e, (pronounced "Oil-er"): The mathematical constant that is the base of the natural logarithm

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \lim_{n \to \infty} (1 + \frac{1}{n})^n$$

e=∑n=0∞1n!=limn→∞(1+1n)n

```
# Euler's number (pronounced "Oil-er")
# The mathematical constant
math.e
```

Logs and Exponents

Doc ID: 6689695

```
math.log10(10)
```

```
math.log10(100)
```

```
math.log10(1000)
```

```
math.log10(1)
```

```
math.log10(0.1)
```

```
math.log10(0)
```

```
math.log10(-1)
```

```
2**3
```

```
# built-in
pow(2,3)
```

```
math.pow(2,3) # Converts arguments to floats
```

The following demonstrating modular exponentiation. Some ways are WAY faster than others...

```
pow(2,3,5)
```

```
(2**3) %5
```

```
pow(100,10000,7)
```

```
(100**10000) % 7
```

```
pow(100, 1000000, 7)
```

```
( 100 ** 1000000 ) % 7
```

Trig Functions

Notice argument to the trig function is measured in radians, not degrees.

```
help(math.sin)
```

```
math.sin(0)
```

If radians the following would be 1:

```
math.sin(90)
```

So, we expect the following to be one:

```
math.sin(math.pi/2)
```

And the following should be 0 (halfway around the unit circle):

Doc ID: 6689695

```
math.sin(math.pi)
```

Is this zero?!!? See the `e-16` at the end? That's pretty close, say within rounding error, of 0. Just be careful that you don't check that it is exactly zero.

```
help(math.isclose)
```

```
print(math.sin(math.pi) == 0)
```

```
print(math.isclose(math.sin(math.pi), 0, abs_tol = 10**-15))
```

```
help(math.degrees)
```

```
help(math.radians)
```

```
math.degrees(math.pi)
```

```
math.radians(45) == math.pi / 4
```

Fun Example:

Save the first 1,000 of the Fibonacci sequence to a list (starting with 1,1 not 0,1).
Iterate over that list and print out how many digits each number is.

```
def fib_list(n = 1000, init = [1,1]):
    '''Returns a list of the first n Fibonacci numbers.'''
    fib_list = init
    for x in range(n-2):
        fib_list.append(fib_list[-1] + fib_list[-2])
    return fib_list
```

```
def fib_lengths(fib_list):
    '''Returns a list containing the number of digits in each fibonacci number.
    It can be calculated this way because the list passed in are integers.
    Use as intended!'''
    return [len(str(int(x))) for x in fib_list]
```

```
a = fib_list()
print(a[:20])
```

```
a_lengths = fib_lengths(a)
print(a_lengths[:20])
```

```
b = fib_list(init = [1.0,1.0])
print(b[:20])
```

```
b_lengths = fib_lengths(b)
print(b_lengths[:20])
```

```
def fib_lengths(fib_list):
  '''Returns a list containing the number of digits in each fibonacci number.'''
  return [1 + math.floor(math.log10(x)) for x in fib_list]
```

```
b_lengths = fib_lengths(b)
print(b_lengths[:20])
```

NumPy

Pronounced "Num - Py"... not like stumpy...

If running through labbench, run the next two lines. If running on jupyter-notebook through Anaconda, you can just import numpy.

```
import ipydeps
```

```
ipydeps.pip('numpy')
```

Now we can import numpy:

```
import numpy as np
```

numpy's main object is called `ndarray`. It is:

- homogeneous
- multidimensional
- array

Meaning, it is a table of elements, usually numbers. All elements are the same type. Elements are accessed by a tuple of positive integers.

Dimensions are called axes.
The number of axes is the rank.

```
t = np.array([1,2,1])
```

```
t
```

```
np.ndim(t) # used to be np.rank, but this is deprecated
```

```
a = np.array(np.arange(15).reshape(3,5))
```

```
a
```

```
np.ndim(a)
```

```
a.shape
```

```
a.ndim
```

```
a.dtype.name
```

```
# size in bytes ... like sizeof operator, but easier to get to.
a.itemsize
```

```
a.size
```

```
type(a)
```

Doc ID: 6689695

We have three ways to access "row" 1 of a:

`a[1] a[1,] a[1,:]`

```
a[1]
```

To get column in position 2:

```
a[:,2]
```

To get single element "row" 0, column 2:

```
a[0,2]
```

Creating and modifying array:

```
c = np.array([2,3,4])
```

```
c.dtype.name
```

```
d = np.array([1.2,3.5,5.1])
```

```
d.dtype.name
```

Let's change one element of c:

```
c[1] = 4.5
```

```
c.dtype.name
```

Ut oh, the type didn't change. But we tried to add a float! Let's see what happened:

```
c
```

The array was updated, but the value we were adding was converted to an `int64`. Be careful. The type matters!!

This doesn't work:

```
f = np.array(1,2,3,4)
```

Needs to be this:

```
f = np.array([1,2,3,4])
```

Interprets a sequence of sequences as a two dimensional array. Sequence of sequence of sequences as a three dimenional array... you get the pattern?

```
g = np.array([[1.5,2,3],[4,5,6]])
g
```

Type can be specified at creation time:

```
h = np.array([[1,2],[3,4]], dtype = complex)
print(h)
print()
print(h.dtype.name)
```

Remember c? Here is how we can change it from integers to floats:

Doc ID: 6689695

```
print(c)
c = np.array(c, dtype = float)
print(c)
c[1] = 4.5
print(c)
```

Suppose you know what size you want, but you want it to have all zeros or ones and you don't want to write them all out. Notice there is one parameter we are passing for the dimensions, but it is a tuple.

```
np.zeros((3,4))
```

```
np.zeros((3,4), dtype = int)
```

```
np.ones((2,3,4), dtype = np.int16)
```

The following is FAST, but dangerous. It does not initialize the entries in the array, but takes whatever is in memory. USE WITH CAUTION.

```
np.empty((2,3))
```

You know range? Well, there is arange. Which allows us to create an array containing a range of numbers evenly spaced.

```
np.arange(10,30,5) # Start, stop, step
```

```
np.arange(0.2,0.3, 0.01) # Can do floats!!
```

Say we don't want to do the math to see what our step should be but we know how many numbers we want... that's what linspace is for! We get evenly spaced samples calculated over the interval. It takes a start, stop and the number of samples you want. There are other optional arguments, but you can figure those out!

```
np.linspace(0,2,9) # 9 numbers evenly spaced numbers between 0 and 2, INCLUSIVE
```

Playing wth "matrices" ... or are they? The following are **not** necessarily the results of matrix operations. What exactly is going on here?

```
a
```

```
a + 1
```

```
a / 2
```

```
a // 2
```

```
pow(2,a)
```

```
b = a // 2
```

```
a + b
```

```
a * b # Not matrix multiplication
```

```
b.T  # Transpose
```

Matplotlib

Doc ID: 6689695

```
## Only if you are on labbench. If using anaconda you don't need to do this.
## If you haven't already improt ipydeps, uncomment the next line also before running
# import ipydeps
ipydeps.pip('matplotlib')
```

```
import matplotlib.pyplot as plt
## If you haven't imported numpy, do so!
#import numpy as np
```

Line Plot

```
a = np.linspace(0,10,100)
b = np.exp(-a)
plt.plot(a,b)
plt.show()
```

Histogram

```
from numpy.random import normal,rand
x = normal(size = 200)
plt.hist(x,bins = 30)
plt.show()
```

3D Plot

```
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm)
plt.show()
```

```
plt.plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plt.plot([1,2,3], [1,4,9], 'rs',  label='line 2')
plt.axis([-2, 5, -2, 10])
plt.plot(a, b, color='green', linestyle='dashed', marker='o', markerfacecolor='blue', markersize=12)
plt.show()
```

Doc ID: 6689695

Sage

To be filled in...

Doc ID: 6689695

COMP3321: Math, Visualization, and More!

(b)(3)-P.L. 86-36

0

Updated almost 3 years ago by [redacted] in COMP 3321

3 5 473 111

fcs6 classes matplotlib numpy python

(U) Python Math, Visualization, and More!

Recommendations

Python Math, Visualization, and More!

This notebook will give a very basic overview to some mathematical and scientific packages in Python, as well as some tools to visualize data.

Credit:

Much of this material is developed by Continuum Analytics, based on a tutorial created by R.R. Johansson.

What is NumPy?

Numpy is a Python library that provides multi-dimensional arrays, matrices, and fast operations on these data structures.

NumPy arrays have:

- fixed size
  - all elements have the same type
    - that type may be compound and/or user-defined
  - fast operations from:
    - vectorization — implicit looping
    - pre-compiled C code using high-quality libraries
      - NumPy default
      - BLAS/ATLAS
      - Intel's MKL

Doc ID: 6689695

NumPy's Uses and Capabilities

- Image and signal processing
- Linear algebra
- Data transformation and query
- Time series analysis
- Statistical analysis

Numpy Ecosystem


```
# Run this if on LABBENCH
import ipydeps

packages = ['matplotlib', 'numpy']
for i in packages:
  ipydeps.pip(i)
```

Let's install necesary packages

```
import numpy as np
import matplotlib as mpl
import numpy.random as npr
vsep = "\n--------------------\n"
```

`matplotlib` — 2D and 3D plotting in Python

```
# This line configures matplotlib to show figures embedded in the notebook,
# instead of opening a new window for each figure. More about that later.
# If you are using an old version of IPython, try using '%pylab inline' instead.
%matplotlib inline
```

Introduction

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

Matplotlib is well suited for generating figures for scientific publications because all aspects of the figure can be controlled *programmatically*. This is important for reproducibility, and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page: http://matplotlib.org/

To get started using Matplotlib in a Python program, import the matplotlib.pyplot module under the name plt:

```
import matplotlib.pyplot as plt
```

The matplotlib MATLAB-like API

A great way to get started with plotting using matplotlib is to use the MATLAB-like API provided by matplotlib.
It is designed to be compatible with MATLAB's plotting functions, so if you are familiar with MATLAB, start here.

```
x = np.linspace(0, 5, 100)
y = x ** 2
```

```
x, y
```

```
plt.figure()
plt.plot(x, y, 'g')
plt.xlabel('x')
plt.ylabel('y')
plt.title('title')
plt.show()
```

```
plt.subplot(1,2,1)
plt.plot(x, y, 'r--')
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```

The matplotlib object-oriented API

The main idea with object-oriented programming is to have objects to which one can apply functions and actions, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

To use the object-oriented API, we start out very much like in the previous example, but instead of creating a new global figure instance, we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

```
fig = plt.figure()

graph = fig.add_axes([0, 0, 1, 0.3]) # left, bottom, width, height (range 0 to 1)

graph.plot(x, y, 'r')

graph.set_xlabel('x')
graph.set_ylabel('y')
graph.set_title('title');
```

Ithough a bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

Doc ID: 6689695

```
fig = plt.figure()

graph1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
graph2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
graph1.plot(x, y, 'r')
graph1.set_xlabel('x')
graph1.set_ylabel('y')
graph1.set_title('Title\n')

# insert
graph2.plot(y, x, 'g')
graph2.set_xlabel('y')
graph2.set_ylabel('x')
graph2.set_title('Inset Title');
```

To save a figure to a file, we can use the savefig method in the Figure class:

```
fig.savefig("filename.png")
```

seaborn — statistical data visualization

Seaborn is a Python visualization library based on `matplotlib`. It provides a high-level interface for drawing attractive statistical graphics.

Homepage for the Seaborn project: http://stanford.edu/~mwaskom/software/seaborn/

bokeh — web-based interactive visualization

Bokeh is a Python interactive visualization library that targets modern web browsers for presentation. Its goal is to provide elegant, concise construction of novel graphics in the style of D3.js, but also deliver this capability with high-performance interactivity over very large or streaming datasets. Bokeh can help anyone who would like to quickly and easily create interactive plots, dashboards, and data applications.

Homepage for Bokeh: http://bokeh.pydata.org/

NumPy Arrays

NumPy arrays (`numpy.ndarray`) are the fundamental data type in NumPy. They have:

- shape
- an element type called *dtype*

For example:

```
M, N = 5, 8
arr = np.zeros(shape=(M,N), dtype=float)
arr
```

Doc ID: 6689695

```
arrzeros = np.zeros(30)
print(arrzeros.dtype, arrzeros.shape)
arrzeros
```

```
arrzeros2 = np.zeros((30,2))
print(arrzeros2.dtype, arrzeros2.shape)
arrzeros2
```

is the NumPy array corresponding to the two-dimensional matrix:

NumPy has both a general N-dimensional array *and* a specific 2-dimensional matrix data type. NumPy arrays may have an arbitrary number of dimensions.

NumPy arrays support vectorized mathematical operation.

```
arr = np.arange(15).reshape(3,5)
print("original:")
print(arr)

print()

print("elementwise computed:")
print(((arr+4)*2) % 30)
print(arr.dtype)
```

```
 ((arr.reshape(15,)+4)*2) % 30
```

```
((arr.reshape(5,3)+4)*2) % 30
```

```
arr.reshape(4, 7)
```

Array Shape

Many array creation functions take a shape parameter. For a 1D array, the shape can be an integer.

```
print(np.zeros(shape=5, dtype=np.float16))
```

For nD arrays, the shape needs to be given as a tuple.

```
print(np.zeros(shape=(4,3,2), dtype=float), end=vsep)
print(repr(np.zeros(shape=(4,3,2), dtype=float)))
```

```
print(np.zeros(shape=(2,2,2,2)))
```

Array Types

All arrays have a specific type for their associated elements. Every element in the array shares that type. The NumPy terminology for this type is *dtype*. The basic types are *bool*, *int*, *uint*, *float*, and *complex*. The types may be modified by a number indicating their size in bits. Python's built-in types can be used as a corresponding dtype. Note, the generic NumPy types end with an underscore ("_") to differentiate the name from the Python built-in.

Doc ID: 6689695

Python Type	NumPy dtype
bool	np.bool_
int	np.int_
float	np.float_
complex	np.complex_

Here is one example of specifying a *dtype*:

```
arr = np.zeros(shape=(5,), dtype=np.float_) # NumPy default sized float
print(arr, "->", arr.dtype)
```

Watch out, though!

```
array = np.array([4192984799048971232, 3, 4], dtype=np.int16)
array
```

```
np.float16('nan')
```

Check the default type...

```
np.array([1], dtype=int).dtype
```

Now we'll take just a moment to define one quick helper function to show us these details in a pretty format.

```
def dump_array(arr):
    print("%s array of %s:" % (arr.shape, arr.dtype))
    print(arr)

vsep = "\n--------------------\n"
```

Array Creation

NumPy provides a number of ways to create an array.

np.zeros and np.ones

```
zrr = np.zeros(shape=(2,3))
dump_array(zrr)
```

```
print(np.ones(shape=(2,5)))
```

```
one_arr = np.ones(shape = (2,2), dtype=int)
dump_array(one_arr)
```

np.empty

`np.empty` is lightning quick because it simply requests some amount of memory from the operating system and then *does nothing with it*. Thus, the array returned by `np.empty` is *uninitialized*. Consider yourself warned. `np.empty` is very useful if you know you are going to fill up all the (used) elements of your array later.

```
# DANGER!  uninitialized array
# (re-run this cell and you will very likely see different values)
err = np.empty(shape=(2,3), dtype=int)
dump_array(err)
```

np.arange

`np.arange` generates sequences of numbers like Python's `range` built-in. Non-integer step values may lead to unexpected results; for these cases, you may prefer `np.linspace` and see below. (For a quick — and mostly practical — discussion of the perils of floating-point approximations, see https://docs.python.org/2/tutorial/floatingpoint.html).

- a single value is a stopping point
- two values are a starting point and a stopping point
- three values are a start, a stop, and a step size

As with `range`, the ending point is *not* included.

```
print("int arg: %s" % np.arange(10), end=vsep)     # cf. range(stop)
print("float arg: %s" % np.arange(10.0), end=vsep) # cf. range(stop)
print("step: %s" % np.arange(0, 12, 2), end=vsep)  # end point excluded
print("neg. step: %s" % np.arange(10, 0, -1.0))
```

np.linspace

`np.linspace(BEGIN, END, NUMPT)` generates exactly *NUMPT* number of points, evenly spaced, on $[\mathrm{BEGIN}, \mathrm{END}]$ [BEGIN,END]. Unlike Python's `range` and `np.arange`, this function is inclusive at BEGIN and END (it produces a closed interval).

```
print("End-points are included:", end=vsep)
print(np.linspace(0, 10, 2), end=vsep)
print(np.linspace(0, 10, 3), end=vsep)
print(np.linspace(0, 10, 4), end=vsep)
print(np.linspace(0, 10, 20), end=vsep)
```

Diagonal arrays: np.eye and np.diag

`np.eye(N)` produces an array with shape (N,N) and ones on the diagonal (an NxN identity matrix).

```
print(np.eye(3))
```

Arrays from Random Distributions

It is common to create arrays whose elements are samples from a random distribution. For the many options, see:

- help(np.random)
- scipy

Uniform on [0,1)

```
print("Uniform on [0,1):")
dump_array(npr.random((2,5)))
```

Standard Normal

`np.random` has some redundancy. It also has some variation in calling conventions.

- `standard_normal` takes *one* tuple argument
- `randn` (which is very common to see in code) takes n arguments where n is the number of dimensions in the result

```
print("std. normal - N(0,1):")
dump_array(npr.standard_normal((2,5)))
print(vsep)
dump_array(npr.randn(2,5)) # one tuple parameter
```

Arrays From a Python List...and a warning!

It is also possible to create NumPy arrays from Python lists and tuples. While this is a nice capability, remember that instantiating a Python list can take relatively long compared to directly using NumPy building blocks. Other containers and iterables will not, generally, give useful results.

```
dump_array(np.array([1, 2, 3]))

print()
dump_array(np.array([10.0, 20.0, 3]))
```

Dimensionality is maintained within nested lists:

```
dump_array(np.array([[1, 2, 3],
                     [4, 5, 6]]))

print()
dump_array(np.array([[1.0, 2],
                     [3, 4],
                     [5, 6]]))
```

Accessing Array Items

Indexing

Items in NumPy arrays may be accessed using a single index composed of multiple values

Doc ID: 6689695

```
arr = np.arange(24).reshape(4,6) # random.randint(11, size=(4, 6))

print("the array:")
print(arr, end=vsep)

print("index [3,2] :", arr[3,2], end=vsep)
print("index [3]   :", arr[3], end=vsep)

# non-idiomatic, creates a view of arr[3] then indexes into that copy
print("index [3][2]:", arr[3][2])
```

Compare this with indexing into a nested Python list

```
aList = [list(row) for row in arr]
print(aList)
print(aList[3][2])

try:
    print(aList[3,2])
except TypeError as e:
    print("Unhappy with multi-value index")
    print("Exception message:", e)
```

Slicing

We can also use slicing to select entire row and columns at once:

![default](mef_numpy_slice_01-noalpha.png) ![default](mef_numpy_slice_02-noalpha.png)

Important Differences Between Python Slicing and NumPy Slicing

- Python slicing returns a **copy** of the original data
  - Changing the slice won't change the original.
- NumPy slicing returns a view of the original data
  - Changing the slice **will** change the original data

The NumPy Indexing Page has a lot more information.

```
print("array:")
print(arr)

print("\naccessing a row:")
dump_array(arr[2,:])

print("\naccessing a column:")
dump_array(arr[:,2])

print("\na row:", arr[2,:], "has shape:", arr[2,:].shape)
print("\na col:", arr[:,2], "has shape:", arr[:,2].shape)
```

Bear in mind that numerical indexing will reduce the dimensionality of the array. Slicing from `index` to `index+1` can be used to keep that dimension if you need it.

```
print("lost dimension:", end=' ')
dump_array(arr[2, 1:4])

print("\nkept dimension:", end=' ')
dump_array(arr[2:3, 1:4])
```

Region Selection and Assignment

Multiple slices, as part of an index, can select a region out of an array

```
print("array:")
print(arr)

print("\na sub-array:")
dump_array(arr[1:3, 2:4])
```

Slices are always views of the underlying array. Thus, modifying them modifies the underlying array

```
arr = np.arange(24).reshape(4,6)
print("even elements (at odd indices) of first row:")
print(arr[0, ::2]) # select every other element from first row

arr[0,::2] = -1   # update is done in-place, no copy

print("\nafter assinging to those:")
print(arr)
```

Working with Arrays

Math is quite simple—and this is part of the reason that using NumPy arrays can significantly simplify numerical code. The generic pattern array OP scalar (or scalar OP array), applies OP (with the scalar value) across elements of array.

Doc ID: 6689695

```
# array OP scalar applies across all elements and creates a new array
arr = np.arange(10)
print("     arr:", arr)
print(" arr + 1:", arr + 1)
print(" arr * 2:", arr * 2)
print("arr ** 2:", arr ** 2)
print("2 ** arr:", 2 ** arr)

# bit-wise ops (cf. np.logical_and, etc.)
print(" arr | 1:", arr | 1)
print(" arr & 1:", arr & 1)

# NOTE:  arr += 1, etc. for in-place
```

```
# array OP array works element-by-element and creates a new array
arr1 = np.arange(5)
arr2 = 2 ** arr1 # makes a new array

print(arr1, "+", arr2, "=", arr1 + arr2, end=vsep)
print(arr1, "*", arr2, "=", arr1 * arr2)
```

Elementwise vs. matrix multiplications

NumPy arrays and matrices are related, but slightly different types.

```
a, b = np.arange(8).reshape(2,4), np.arange(10,18).reshape(2,4)
print("a")
print(a)
print("b")
print(b, end=vsep)
print("Elementwise multiplication: a * b")
print(a * b, end=vsep)
print("Dot product: np.dot(a.T, b)")
print(np.dot(a.T, b), end=vsep)
print("Dot product as an array method: a.T.dot(b)")
print(a.T.dot(b), end=vsep)

amat, bmat = np.matrix(a), np.matrix(b)
print("amat, bmat = np.matrix(a), np.matrix(b)")
print('amat')
print(amat)
print('bmat')
print(bmat, end=vsep)
print("Dot product of matrices: amat.T * bmat")
print(amat.T * bmat, end=vsep)
print("Dot product in Python 3.5+: a.T @ b")
print(amat.T @ bmat)
```

Some Additional NumPy Subpackages

- `np.fft` — Fast Fourier transforms
- `np.polynomial` — Orthogonal polynomials, spline fitting
- `np.linalg` — Linear algebra
- `cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd`
- `np.math` — C standard library math functions
- `np.random` — Random number generation
- `beta, gamma, geometric, hypergeometric, lognormal, normal, poisson, uniform, weibull`
- many others, if you need it, NumPy probably has it.

FFT

```
PI = np.pi
t = np.linspace(0, 120, 4000)
nrr = np.random.random

signal  =  12 * np.sin(3 * 2*PI*t)   # 3 Hz
signal +=   6 * np.sin(8 * 2*PI*t)   # 8 Hz
signal += 1.5 * nrr(len(t))          # noise
```

```
# General FFT calculation
FFT   = abs(np.fft.fft(signal))
freqs = np.fft.fftfreq(signal.size, t[1] - t[0])
plt.plot(t, signal); plt.xlim(0, 4); plt.show()
plt.plot(freqs, FFT);
```

```
# For one-dimensional real inputs we can discard the negative frequencies
FFT   = abs(np.fft.rfft(signal))
freqs = np.fft.rfftfreq(signal.size, t[1] - t[0])
plt.plot(freqs, FFT); plt.xlim(-0.2, 10);
```

Testing speedup of discarding negative frequencies

```
%%timeit
FFT   = abs(np.fft.fft(signal))
freqs = np.fft.fftfreq(signal.size, t[1] - t[0])
```

```
%%timeit
FFT   = abs(np.fft.rfft(signal))
freqs = np.fft.rfftfreq(signal.size, t[1] - t[0])
```

SciPy - Library of scientific algorithms for Python

The SciPy framework builds on top of the low-level NumPy framework for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are:

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module...or only import the subpackages we need...

Fourier transform

Fourier transforms are one of the universal tools in computational physics; they appear over and over again in different contexts. SciPy provides functions for accessing the classic FFTPACK library from NetLib, an efficient and well tested FFT library written in FORTRAN. The SciPy API has a few additional convenience functions, but overall the API is closely related to the original FORTRAN library.

To use the `fftpack` module in a python program, include it using:

```
import scipy.fftpack as spfft
```

```
# General FFT calculation
FFT   = abs(spfft.fft(signal))
freqs = spfft.fftfreq(signal.size, t[1] - t[0])
plt.plot(t, signal); plt.xlim(0, 4); plt.show()
plt.plot(freqs, FFT);
```

NumPy FFT vs. SciPy FFT vs. FFTW vs. MKLFFT

Which FFT library should you use?
If you want to use the MKL, you must use NumPy.
The default installations of NumPy and SciPy use FFTPACK
FFTW is faster than FFTPACK, and often faster than MKL

Installing PyFFTW

1. Install FFTW

- apt-get install libfftw3-3 libfftw3-dev
- yum install fftw-devel

1. pip install pyfftw

Doc ID: 6689695

Interpolation

Interpolation is simple and convenient in SciPy: The `interp1d` function, when given arrays describing X and Y data, returns an object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X). It returns the corresponding interpolated y value:

```
import scipy.interpolate as spinter
```

```
def f(x):
    return np.sin(x)
```

```
n = np.arange(0, 10)
x = np.linspace(0, 9, 100)

y_meas = f(n) + 0.1 * np.random.randn(len(n)) # simulate measurement with noise
y_real = f(x)

linear_interpolation = spinter.interp1d(n, y_meas)
y_interp1 = linear_interpolation(x)

cubic_interpolation = spinter.interp1d(n, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
```

```
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(n, y_meas, 'bs', label='noisy data')
ax.plot(x, y_real, 'k', lw=2, label='true function')
ax.plot(x, y_interp1, 'r', label='linear interp')
ax.plot(x, y_interp2, 'g', label='cubic interp')
ax.legend(loc=3);
```

Doc ID: 6689695

COMP3321 (U) Python Visualization

(b)(3)-P.L. 86-36

0

Updated over 1 year ago by [redacted] in COMP 3321

3 13 631 246

fcx91 matplotlib bokeh seaborn visualization holoviews

(U) This notebook gives an overview of three visualization methods within Python, matplotlib, seaborn, and bokeh.

Recommendations

Python Visualization

This notebook will give a basic overview of some tools to visualize data. Much of this material is developed by Continuum Analytics, based on a tutorial created by Wesley Emeneker.

First, install necessary packages:

```
# Run this if on LABBENCH
import ipydeps

packages = ['matplotlib', 'seaborn', 'bokeh' ,'pandas', 'numpy', 'scipy',
            'holoviews']
ipydeps.pip(packages)

import numpy as np
import pandas as pd
```

Visualization Choices

There are many different visualization chioces within Python. In this notebook we will look at three main options:

- Matplotlib

- Seaborn
- Bokeh

Each of these options are built with slightly different purposes in mind, so choose the option which best suits your needs!

Matplotlib

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

Matplotlib is well suited for generating figures for scientific publications because all aspects of the figure can be controlled *programmatically*. This is important for reproducibility, and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page.

To get started using Matplotlib in a Python program, import the matplotlib.pyplot module under the name plt:

```
import matplotlib.pyplot as plt
# This line configures matplotlib to show figures embedded
# in the notebook, instead of opening a new window for each
# figure. More about that later.  If you are using an old
# version of IPython, try using '%pylab inline' instead.
%matplotlib inline
```

The `matplotlib` MATLAB-like API

A great way to get started with plotting using matplotlib is to use the MATLAB-like API provided by matplotlib. It is designed to be compatible with MATLAB's plotting functions, so if you are familiar with MATLAB, start here.

```
x = np.linspace(0, 5, 100)
y = x ** 2
print(x[0:10])
print(y[0:10])
```

```
plt.figure()
plt.plot(x, y, 'g')
plt.xlabel('x')
plt.ylabel('y')
plt.title('title')
plt.show()
```

```
plt.subplot(1,2,1)
plt.plot(x, y, 'r--')
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```

The `matplotlib` object-oriented API

The main idea with object-oriented programming is to have objects to which one can apply functions and actions, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

To use the object-oriented API, we start out very much like in the previous example, but instead of creating a new global figure instance, we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

```
fig = plt.figure()

graph = fig.add_axes([0, 0, 1, 0.3]) # left, bottom, width, height (range 0 to 1)

graph.plot(x, y, 'r')

graph.set_xlabel('x')
graph.set_ylabel('y')
graph.set_title('title');
```

Although a bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

```
fig = plt.figure()

graph1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
graph2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
graph1.plot(x, y, 'r')
graph1.set_xlabel('x')
graph1.set_ylabel('y')
graph1.set_title('Title\n')

# insert
graph2.plot(y, x, 'g')
graph2.set_xlabel('y')
graph2.set_ylabel('x')
graph2.set_title('Inset Title');
```

Plotting categorical data

Note: this works in matplotlib 2.0.0; in version 2.1, you can enter the categorical data directly on many of the matplotlib plotting methods.

Doc ID: 6689695

```
# initialize our data here, turning this into a list of names and a list of counts
data ={'apples': 10, 'oranges': 15, 'lemons': 5, 'limes': 20}
names = list(data.keys())
values = list(data.values())

# first have to create numeric values to cover the axis with the categorical data
N = len(names)
ind = np.arange(N)
width = 0.35

# this will make three separate plots to demonstrate
fig, axs = plt.subplots(1, 3, figsize=(15, 3), sharey=True)
axs[0].bar(ind + width, values)
axs[1].scatter(ind + width, values)
axs[2].plot(ind + width, values)

# here we'll space out the tick marks appropriately and replace the numbers with the names
# for the labels
for ax in axs:
  ax.set_xticks(ind + width)
  ax.set_xticklabels(names)

fig.suptitle("Categorical Plotting")
plt.show(fig)
```

In Matplotlib 2.1+, we can do this directly

```
from bokeh.sampledata.autompg import autompg as df

fig = plt.figure()

graph = fig.add_axes([0.1, 0.1, 2.0, 0.8])

num_vehicles = 8

graph.bar(
  df['name'].value_counts().index[:num_vehicles],
  df['name'].value_counts().values[:num_vehicles]
)

graph.set_title('Number of vehicles')
graph.set_xlabel('Vehicle name')
graph.set_ylabel('Count')

plt.show()
```

To save a figure to a file, we can use the savefig method in the Figure class:

```
fig.savefig("filename.png")
```

The real power of Matplotlib comes with ploting of numerical data, and so we will wait to delve into Matplotlib further until we talk more about mathematics in Python.

seaborn — statistical data visualization

Seaborn is a Python visualization library based on `matplotlib`. It provides a high-level interface for drawing attractive statistical graphics.

The homepage for the Seaborn project on the internet is here.

```
import seaborn as sns
import pandas as pd
sns.set()
```

```
fig = plt.figure()

graph1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
graph2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
graph1.plot(x, y, 'r')
graph1.set_xlabel('x')
graph1.set_ylabel('y')
graph1.set_title('Title\n')

# insert
graph2.plot(y, x, 'g')
graph2.set_xlabel('y')
graph2.set_ylabel('x')
graph2.set_title('Inset Title');
```

More examples!

```
import random
df = pd.DataFrame()
df['x'] = random.sample(range(1,100),25)
df['y'] = random.sample(range(1,100),25)
df.head()
```

Scatterplot

```
sns.lmplot('x','y',data=df,fit_reg=False);
```

Density Plot

```
sns.kdeplot(df.y);
```

Contour plots

```
sns.kdeplot(df.y, df.x);
```

Distribution plots

```
sns.distplot(df.x);
```

Histogram

```
plt.hist(df.x,alpha=.3)
sns.rugplot(df.x);
```

Heatmaps

```
sns.heatmap([df.y, df.x], annot=True, fmt="d");
```

Bokeh

Bokeh is a Python interactive visualization library whose goal is to provide elegant graphics in the style of D3.js while maintaining high-performance interactivity over large or streaming datasets. Bokeh is designed to generate web-based interactive plots, and as such, it may not be able to provide as fine a resolution as Matplotlib. The homepage for the Bokeh project on the internet is here.

There are multiple options for displaying Bokeh graphics. The two most common methods are `output_file()` and `output_notebook()`:

- The `output_notebook()` method works with `show()` to display the plot within a Jupyter notebook.
- The `output_file()` method works with `save()` to generate a static HTML file. The data is saved with the plot to the HTML file.

In this notebook, we will focus on `output_notebook()`.

```
from bokeh.plotting import figure, output_notebook, show

from bokeh.resources import INLINE
output_notebook(resources=INLINE)
```

```
import holoviews as hv
hv.extension('bokeh')
```

First, we will make a Bokeh plot of a `line`. The `line` function takes a list of x and y coordinates as input.

```
# set up some data
import numpy as np

x = np.linspace(0, 4*np.pi, 100)
y = np.sin(x)

#plot a line
plot = figure()
plot.line(x, y)
show(plot)
```

Styling and Appearance

The 'line' above is an example of an object called 'Glyph'. Glyphs are made of 'lines' and 'filled areas'. Style arguments can be passed to any glyph as keywords. Some properties include:

- Line properties: `line_color` , `line_alpha` , `line_width` , and `line_dash` .
- Fill properties: `fill_color` and `fill_alpha` .

Bokeh uses CSS Color Names.

Here is another example showing styling options:

```
x = np.linspace(0, 4*np.pi, 100)
y = np.sin(x)


plot = figure(title="Sine Function")
plot.xaxis.axis_label='x'
plot.yaxis.axis_label='amplitude'


plot.line(x, y,
  line_color='blue',
  line_width=2,
  legend='sin(x)')

plot.circle(x, 2*y,
  fill_color='red',
  line_color='black',
  fill_alpha=0.2,
  size=10,
  legend='2sin(x)')

#Line_dash is an aribrary length list of lengths
#alternating in [color, blank, color, ...]
plot.line(x, np.sin(2*x),
  line_color='green',
  line_dash=[10,5,2,5],
  line_width=2,
  legend='sin(2x)')

show(plot)
```

Charts

Bar charts

The Bar high-level chart can produce bar charts in various styles. Bar charts are configured with a DataFrame data object, and a column to group. This column will label the x-axis range. Each group is aggregated over the values column and bars are show for the totals:

```
from bokeh.sampledata.autompg import autompg as autompg
autompg.head()
```

```
hp_by_cyl = autompg.groupby('cyl', as_index=False).agg({'hp': np.mean})
p = figure(title="Average HP by CYL", plot_width=600, plot_height=400)
p.vbar(x='cyl', top='hp', width=0.5, source=hp_by_cyl)
show(p)
```

Categorical Bar Chart

For a categorical bar chart, we still use `p.vbar` as above, but the top value will be the counts for the items in `x`. For the below example, we used the `ColumnDataSource` class from `bokeh.models` to actually store the data, which we can then pass in `p.vbar` under the `source` keyword. We also imported a color palette from `bokeh.palettes` to use as the color palette, passed in with the `color` keyword in our `ColumnDataSource`. (Note: when using color palettes, you need to make sure there are enough colors in the palette to cover all the values in your data.)

```
from bokeh.models import ColumnDataSource
from bokeh.palettes import Spectral6

fruits = ['Apples','Pears','Nectarines','Plums','Grapes','Strawberries']
counts = [5,3,4,2,4,6]

source = ColumnDataSource(data=dict(fruits=fruits, counts=counts,
                                    color=Spectral6))

p = figure(x_range=fruits, y_range=(0,max(counts)+3), plot_height=250,
           title="Fruit Counts", toolbar_location=None, tools="")

p.vbar(x='fruits', top='counts', width=0.9, color='color',
       legend='fruits', source=source)

p.xgrid.grid_line_color = None
p.legend.orientation = 'horizontal'
p.legend.location = 'top_center'

show(p)
```

Histograms

Simple histogram using Holoviews

Using `holoviews`, we can easily create a histogram on top of Bokeh with the `hv.Histogram` function.

```
%%output size=150

%%opts Histogram (fill_color='#CD5C5C', line_color='black')

hist = hv.Histogram(np.histogram(autompg['mpg'], bins=20), kdims=['mpg'], extents=(7, 0, 48, 45))

hist
```

More complicated histogram using native Bokeh syntax

We can also use the `quad` method. In this example below, we're actually using the `np.histogram` function to create our histogram values, which are then passed into the `quad` method to create the histogram.

```
import scipy.special
import numpy as np
from bokeh.layouts import gridplot

p = figure(title="Normal Distribution (mu=0, sigma=0.5)", tools="save",
           background_fill_color="#E8DDCB")

mu, sigma = 0, 0.5

measured = np.random.normal(mu, sigma, 1000)
hist, edges = np.histogram(measured, density=True, bins=50)

x = np.linspace(-2, 2, 1000)

p.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:],
       fill_color="#036564", line_color="#033649")
p.xaxis.axis_label='x'
p.yaxis.axis_label='Pr(x)'

show(p)
```

Scatter plots

```
%%output size=150

scatter = hv.Scatter(autompg.loc[:, ['mpg', 'hp']])

scatter
```

Curves

```
%%output size=150

accel_by_hp = autompg.groupby('hp', as_index=False).agg({'accel': np.mean})

%opts Curve [height=200, width=400, tools=['hover']]
%opts Curve (color='red', line_width=1.5)

curve = hv.Curve(accel_by_hp)

curve
```

Spikes

```
%%output size=150

spikes = hv.Spikes(accel_by_hp)

spikes
```

Using layouts to combine plots

As simple as using + to add plots together

```
%%output size=120

%%opts Curve [height=200, width=400, xaxis='bottom']
%%opts Curve (color='red', line_width=1.5)

%%opts Spikes [height=200, width=400, yaxis='left']
%%opts Spikes (color='black', line_width=0.8)

layout = curve + spikes

layout
```

A taste of advanced Bokeh features

Bokeh is loaded with wonderful features. Here are two final examples with no explanation. See thesenotebooks for additional Bokeh information.

```
from bokeh.sampledata.iris import flowers
flowers.head()
```

```
from bokeh.models import BoxZoomTool,ResetTool,HoverTool

## Add a new Series mapping the species to a color
colormap = {'setosa': 'red', 'versicolor': 'green', 'virginica': 'blue'}
flowers['color'] = flowers['species'].map(lambda x: colormap[x])

tools = [BoxZoomTool(),ResetTool(),HoverTool()]

plot = figure(title = "Iris Morphology", tools=tools)
plot.xaxis.axis_label = 'Petal Length'
plot.yaxis.axis_label = 'Petal Width'

plot.circle(
    flowers["petal_length"],
    flowers["petal_width"],
    color=flowers["color"], #assign the color to each circle
    fill_alpha=0.2, size=10 )

show(plot)
```

Doc ID: 6689695

```
x = np.linspace(0, 4*np.pi, 100)
y = np.sin(x)
```

```
plot = figure(tools='reset,box_select,lasso_select,help')
plot.circle(x, y, color='blue')
show(plot)
```

Doc ID: 6689695

(b)(3)-P.L. 86-36

Module: Pandas

0

Updated over 1 year ago by [redacted] in COMP 3321

3 9 689 223

fcx93 pandas python numpy dataframe datascience tutorial

(U) This modules covers the Pandas package in Python, for working with dataframes.

Recommendations

Pandas Resource & Examples

(Note: this was modified from the Pandamonium notebook by [redacted] on nbGallery.)

This resource should help people who are new to Pandas and need to explore capabilities or learn the syntax. I'm going to provide a few examples for each command I introduce. It's important to mention that these are not all the commands available!

If you prefer video tutorials, here's a Safari series =>
Data Analysis with Python and Pandas

Also note that Pandas documentions is available in DevDocs.

First we'll import and install all necessary modules.

```
import ipydeps
modules = ['pandas','xlrd','bokeh','numpy',
           'requests','requests_pki','openpyxl']
ipydeps.pip(modules)
```

"pd" is the standard abbreviation for pandas, and "np" for numpy

```
import math
import pandas as pd
import numpy as np

#This is only included to give us a sample dataframe to work with
from bokeh.sampledata.autompg import autompg as df
```

Creating a DataFrame

The very basics of creating your own DataFrame. I don't find myself creating them from scratch often but I do create empty DataFrames like seen a few times further down in the guide.

```
#Create Empty DataFrame Object
df1 = pd.DataFrame()
```

```
#This is the very basic method, create empty DataFrame but specify 4 columns and their names
#You can also specify datatypes, index, and many other advanced things here
df1 = pd.DataFrame(columns=('Column1','Column2','Column3','Column4'))
```

```
#Create testing DataFrames (a, b, c), always useful for evaluating merge/join/concat/append operations.
a = pd.DataFrame([[1,2,3],[3,4,5]], columns=list('ABC'))
b = pd.DataFrame([[5,2,3],[7,4,5]], columns=list('BDE'))
c = pd.DataFrame([[11,12,13],[17,14,15]], columns=list('XYZ'))
```

```
a
```

```
b
```

```
c
```

Reading from and Writing To Files

Super easy in Pandas

CSV

Let's write our autompg dataframe out to csv first so we have one to work this. Note: if you leave the `index` parameter set to `True`, you'll get an extra column called "Unnamed: 0" in your CSV.

```
df.to_csv("autompg.csv", index=False)
```

Now reading it in is super easy.

```
df1 = pd.read_csv("autompg.csv")
df1.head()
```

If the file contains special encoding (if it's not english for example) you can look up the encoding you need for your language or text and include that when you read the file.

```
df1 = pd.read_csv('autompg.csv', encoding = 'utf-8-sig')
```

You can also specify which columns you'd like to read in (if you'd prefer a subset).

```
df2 = pd.read_csv('autompg.csv', usecols=['name', 'mpg'])
df2.head()
```

If your file is not a csv, and uses alternative seperators, you can specify that when you read it in. Your file does not need to have a ".csv" extension to be read by this function, but should be a text file that represents data.

For Example, if you have a .tsv, or tab-delimited file you can specify that to pandas when reading the file in.

```
df1.to_csv("autompg.tsv", index=False, sep='\t')
df1 = pd.read_csv('autompg.tsv', sep='\t')
df1.head()
```

Chunking on Large CSVs

Often times, when working with very large CSVs you will run into errors. There are a few methods to work around these errors outside of a Help Desk ticket for more memory.

If you don't have enough memory to directly open an entire CSV, as when they start going above 500MB-1GB+, you can *sometimes* alleviate the problem by chunking the in-read (opening them in smaller pieces).

Note: your numeric index will be reset each time.

```
# first we'll create a large DataFrame for an example
large_df = pd.DataFrame()
for i in range(100):
    # ignore_index prevents the index from being reset with each DataFrame added
    large_df = large_df.append(df1, ignore_index=True)
large_df.to_csv("large_file.csv", index=False)

#chunk becomes the temporary dataframe containing the data of that chunk size
for chunk in pd.read_csv('large_file.csv', chunksize=1000):
    print(chunk.head(1))
```

Another chunking variation

If you still need to load a very large CSV into memory for deduplication or other processing reasons, there are ways to do it. This method uses a temporary DataFrame for appending, which gets dumped into a master DataFrame after 200 chunks have been processed. Clearing the temporary DataFrame every 200 chunks reduces memory overhead and improves speed during the append process.

You can improve efficiency by adjusting chunksize and the interval that it dumps data into the master DataFrame. There may be more efficient ways to do this, but this is effective. At the end of the cell, we have a DataFrame `df1` which has all the data that we couldn't read all at once.

Notes: I use `ignore_index` in order to have unique index values, since append will automatically preserve index values.

```
df1 = pd.DataFrame()
df2 = pd.DataFrame()

for counter, chunk in enumerate(pd.read_csv('large_file.csv', chunksize=1000)):
    #Every 200 chunks, append df2 to df1, clear memory, start an empty df2
    if (counter % 200) == 0:
        df2 = df2.append(chunk, ignore_index=True)
        df1 = df1.append(df2, ignore_index=True)
        df2 = pd.DataFrame()
    else:
        df2 = df2.append(chunk, ignore_index=True)

#Anything leftover gets appended to master dataframe (df1)
df1 = df1.append(df2, ignore_index=True)

#remove the temporary DataFrame
del df2

print("There are {} rows in this DataFrame.".format(len(df1)))

df1.head()
```

Excel

Use `ExcelWriter` to write a `DataFrame` or multiple `DataFrames` to an Excel workbook.

```
df2 = pd.DataFrame([{'Name': 'Po', 'Occupation': 'Dragon Warrior'},
  {'Name': 'Shifu', 'Occupation': 'Sensei'}])
# this just initializes the workbook
writer = pd.ExcelWriter("test_workbook.xlsx")
# write as many DataFrames as sheets as you want
df1.to_excel(writer, "Sheet1")
df2.to_excel(writer, "Sheet2")
writer.save() # .save() finishes the operation and saves the workbook
```

When reading from an Excel workbook, Pandas assumes you want just the first sheet of the workbook by default.

```
df1 = pd.read_excel('test_workbook.xlsx')
df1.head()
```

To read a specific sheet, simply include the name of the sheet in the read command.

```
df1 = pd.read_excel('test_workbook.xlsx', sheet_name='Sheet2')
df1.head()
```

Loading from JSON/API

This is just a very simple example to show that it's very easy for JSON or API payloads to be converted to a DataFrame, as long as the payload has a structured format that can be interpreted.

Pandas can write a DataFrame to a JSON file, and also read in from a JSON file.

```
df.to_json("json_file.json")

from_json = pd.read_json("json_file.json")

from_json.head()
```

The same can be done for JSON objects instead of files.

```
json_object = df.to_json() # don't specify a file and it will create a JSON object

from_json = pd.read_json(json_object)

from_json.head()
```

DataFrame Information Summaries

Now that your data is imported, we can get down to business.

To retrieve basic infromation about your DataFrame, like the shape (column and row numbers), index values (row identifiers), DataFrame info (attributes of the object), and the count (number of values in the columns).

```
df.shape
```

```
df.index
```

```
df.info()
```

```
df.count()
```

Describe DataFrame

Summary Statistics - DataFrame.describe() will try to process numeric columns by running: (count, mean, standard deviation (std), min, 25%, 50%, 75%, max) output will be that summary.

```
df.describe()
```

Checking Head and Foot of DataFrame

Note: You can use this on most operations (especially in this guide) to get a small preview of the output instead of the entire DataFrame.

```
#Show first 5 rows of DataFrame
df.head()
```

```
#Specify the number of rows to preview
df.head(10)
```

```
#Show last 5 rows of DataFrame
df.tail()
```

```
#Or Specify
df.tail(10)
```

Checking DataTypes

It's important to know how your DataFrame will treat the data contained in specific columns, and how it will read in the columns. Pandas will attempt to automatically parse numbers as int or float, and can be asked to parse dates as datetime objects. Understanding where it succeeded and where an explicit parse statement will be needed is important, the dataframe can provide this information.

Note: Pandas automatically uses numpy objects.

```
#View column names and their associated datatype
df.dtypes
```

```
#Select columns where the datatype is float64 using numpy (a decimal number)
df.select_dtypes([np.float64])
```

```
#Select columns where the datatype is a numpy object (like a string)
df.select_dtypes([np.object])
```

```
#Change the data type of a column
df2 = df.copy()
df2['mpg'] = df2['mpg'].astype(str)
df2['mpg'].unique()
```

Modifying DataFrames

Modifications only work on assignment or when using `inplace=True`, which instructs the `DataFrame` to make the change without reassignment. See examples below.

Change by assignment

```
df2 = df.drop('cyl',axis=1)
df2.head()
```

Change in place

```
df2.drop('hp',axis=1, inplace=True) #inplace
df2.head()
```

View and Rename Columns

Check all column names or Rename specific columns

```
#Check Column Names
df.columns
```

```
#Store column names as a list
x = list(df.columns)
```

Batch renaming columns requires a dictionary of the old values mapped to the new ones.

```
df2 = df.rename(columns={'mpg': 'miles_per_gallon',
                         'cyl': 'cylinders'})
df2.head()
```

Create New Columns

Similar to a dictionary, if a column doesn't exist, this will automatically create it

```
#Will populate entire column with value specified
df2 = df.copy()
df2['year'] = '2017'
df2.head()
```

Accessing Index and Columns

Access a specific column by name or row by index

Change the column placeholders below to actually see working

Columns Available in Practice DataFrame: (mpg, cyl, displ, hp, weight, accel, yr, origin, name)

```
#By Column
df['name'].head()
```

```
#Alternatively and equivalent to above, this won't work if there are spaces in the column name
df.name.head()
```

```
#By Numeric Index, below is specifying 2nd and 3rd rows of values
df.iloc[2:4]
```

```
#By Index + Column
df.loc[[1], ['name']]
```

Remove Duplicates

Important operation for reducing a DataFrame!

Change the column placeholders below to actually see working

Columns Available in Practice DataFrame: (mpg, cyl, displ, hp, weight, accel, yr, origin, name)

```
len(df)
```

```
#first let's create some duplicates
df2 = df.append(df, ignore_index=True)
print("There are {} rows in the DataFrame.".format(len(df2)))

#Remove any rows which contain duplicates of another row
df2.drop_duplicates(inplace=True)
print("There are now {} rows in the DataFrame.".format(len(df2)))
```

```
#or specify columns to reduce the number of cells in a row that must match to be dropped
df2 = df2.drop_duplicates(subset=['mpg'])
print("There are now {} rows in the DataFrame.".format(len(df2)))
```

Filtering on Columns

Filter a DataFrame based on specific column & value parameters. In the example below, we are creating a new DataFrame (df2) from our filter specifications against the sample DataFrame (df).

```
#Created new dataframe where 'cyl' value == 6
df2 = df.loc[df['cyl'] == 6]
df2.head()
```

```
# use reset_index to re-number the index values
df2 = df.loc[df['cyl'] == 6].reset_index(drop=True)
df2.head()
```

```
# not that we don't need .loc for these operations
df2 = df[df['mpg'] >= 16].reset_index(drop=True)
df2.head()
```

Fill or Drop the NaN or null Values

Repair Empty Values or 'NaN' across DataFrame or Columns

Change the column placeholders below to actually see working

Columns Available in Practice DataFrame: (mpg, cyl, displ, hp, weight, accel, yr, origin, name)

Note: `df.dropna` & `df.fillna` are modifications and will modify the sample DataFrame. Remove "inplace=True" from entries to prevent modification

Doc ID: 6689695

```
df.reindex?
```

```
# first we'll add some empty values
df3 = pd.DataFrame([{'name': 'Ford Taurus'}, {'mpg': 18.0}])
df2 = df.append(df3, ignore_index=True)
```

```
#Check for NaN values
df2.loc[df2['name'].isnull()]
```

```
#True/False Output on if columns contain null values
df2.isnull().any()
```

```
#Sum of all missing values by column
df2.isnull().sum()
```

```
#Sum of all missing values across all columns
df2.isnull().sum().sum()
```

```
#Locate all missing values
df2.loc[df2.isnull().T.any()]
```

```
#Fill NaN values
df2.fillna(0).tail()
```

```
#Drop NaN values
df2.dropna().tail()
```

```
#Alternatively target a column
df2['cyl'].fillna(0).tail()
```

```
#Drop row only if all columns are NaN
df2.dropna(how='all').tail()
```

```
#Drop if a specific number of columns are NaN
df2.dropna(thresh=2).tail()
```

```
#Drop if specific columns are NaN
df2.dropna(subset=['displ','hp']).tail()
```

Simple Operations

```
#All Unique values in column
df['mpg'].unique()
```

```
#Count of Unique Values in column
df['cyl'].value_counts()
```

```
#Count of all entries in column
df['hp'].count()
```

```
#sum of all column values
df['hp'].sum()
```

```
#mean of all column values
df['cyl'].mean()
```

```
#median of all column values
df['cyl'].median()
```

```
#min (lowest numeric value) of all column values
df['cyl'].min()
```

```
#max (highest numeric value) of all column values
df['cyl'].max()
```

```
#Standard Deviation of all column values
df['cyl'].std()
```

Sorting Columns

Note: These are just the very basic sort operations. There are many other advanced methods (multi-column sort, index sort, etc) that include multiple arguments.

```
#Sort dataframe by column values
df.sort_values('mpg', ascending=False).head()
```

```
#Multi-Column Sort
df.sort_values(['mpg', 'displ']).head()
```

Merging DataFrames

While many of these are similar, there are specifics and numerous arguments that can be used in conjunction that truly customize the type of DataFrame joining/merging/appending/concating you're trying to accomplish.

Note: I've provided more sample DataFrames (a, b, c) to help illustrate the various methods. Join/Merge act similar to SQL joins. This Wikipedia entry might help but it can take some time to learn and get comfortable with using them all.

```
#example df's
a = pd.DataFrame([[1,2,3],[3,4,5]], columns=list('ABC'))
b = pd.DataFrame([[5,2,3],[7,4,5]], columns=list('BDE'))
c = pd.DataFrame([[11,12,13],[17,14,15]], columns=list('XYZ'))
print(a)
print(b)
print(c)
```

Append DataFrames

Merges 2+ DataFrames, Does not care if dissimilar or similar. Can also use a list of DataFrames.

```
ab = a.append(b)
ab
```

Concatenate DataFrames

Simlar to append, but handles large lists of dataframes well.

```
abc = pd.concat([a,b,c])
abc
```

Join DataFrames

SQL-ish join operations (Inner/Outer etc), can specify join on index, similar columns may require specification

```
joined_df = a.join(b,how='left',lsuffix="_a",rsuffix="_b")
joined_df
```

Merge DataFrames

Merges 2+ DataFrames with overlapping columns, Very similar to join.

```
merged_df = a.merge(b, left_on='B', right_on='D')
merged_df
```

Iterate DataFrame

Iterating is only good for small dataframes, larger dataframes generally require apply/map and functions for efficiency

You will inevitably use these methods at one point or another.

Iter Rows

Access to values is done by index
rows[0] = Index

rows[1] = values as pandas series (similar to a dict)

rows[1][0] = First column value of row, can specify column rows[1]['Column']

```
counter = 0
for row in df.iterrows():
    counter += 1
    if counter > 15:
        break
    print(row[1].keys()[0])
    print(row[1]['name'])
    print(row[0], row[1][0])
```

IterTuples

Faster and more effecient, access to values is slightly different from iterrows (Index is not nested).
rowtuples[0] = Index

rowtuples[1] = First column value

rowtuples[2] = Second column value

```
counter = 0
for rowtuples in df.itertuples():
    counter += 1
    if counter > 15:
        break
    print(rowtuples[1],rowtuples[2],rowtuples[3])
```

Pivoting on DataFrame

Create Excel style pivot tables based on specified criteria

```
#Basic Pivot
df.pivot_table(index=['mpg','name']).head()
```

```
#Specify for a more complex pivot table
df.pivot_table(values=['weight'], index=['cyl','name'], aggfunc=np.mean).head()
```

Boolean Indexing

Filter DataFrame on Multiple Columns and Values using Boolean index

Note: The '&' in this example represents 'and' which might cause confusion. The explanation for this can also be a bit confusing, at least it caught me off guard the first few times. The '&' will create a boolean array (of True/False which is used by the filtering operation to construct the output. When all 3 statements below return true for a row, pandas knows that we want that row in our output. The 'and' comparator functions differently than '&' and will throw a 'the truth value for the array is ambiguous' exception.

```
df.loc[(df['cyl'] < 6) &
       (df['mpg'] > 35)].head()
```

```
# the same thing can be done with .query, for a more SQL-esque way to do it
# just beware that you can run into issues with string formatting when using this method
df.query("cyl < 6 & mpg > 35").head()
```

Crosstab Viewing

Contingency table (also known as a cross tabulation or crosstab) is a type of table in a matrix format that displays the (multivariate) frequency distribution of the variables

```
pd.crosstab(df['cyl'],df['yr'],margins=True)
```

Example using multiple options

Note: This is an example using a combination of techniques seen above. I've also introduced a new method .nlargest

```
#Top Number of Column1 Unique Values based on the Mean of NumColumn Unique Values using .nlargest
df.cyl.value_counts().nlargest(math.ceil(df.mpg.value_counts().mean())).head
```

Create a New Column with simple logic

Useful technique for simple operations

```
#Using .astype(str) I can treat the float64 df['mpg'] column as a string and merge it with other strings
df2 = df.copy()
df2['mpg_str'] = df2['name'] + ' Has MPG ' + df2['mpg'].astype(str)
df2.head()
```

Functions on DataFrames

The fastest and most effecient method of running calculations against an entire dataframe. This will become your new method of 'iterating' over the data and doing analytics.

axis = 0 means function will be applied to each column

axis = 1 means function will be applied to each row

Note: This is a step into more advanced techniques. Map/Apply/Applymap are the most efficient Pandas method of iterating and running functions across a DataFrame.

Map

Map applys a function to each element in a series, very like iterating.

```
def concon(x):
    return 'Adding this String to all values: '+str(x)

df['name'].map(concon).head()
```

Apply

Apply runs a function against the axis specified.
We are creating hp_and_mpg based on results of adding
We are creating a New_Column based on the results of summing Column1 + Column2

```
df2['hp_and_mpg'] = df2[['hp', 'mpg']].apply(sum,axis=1)
df2.loc[:, ['hp', 'mpg', 'hp_and_mpg', 'name']].head()
```

ApplyMap

Runs a function against each element in a dataframe (each 'cell')

```
df.applymap(concon).head()
```

More Function Examples

```
def num_missing(x):
    return sum(x.isnull())

#Check how many missing values in each column
df.apply(num_missing, axis=0)
```

```
#Check how many missing values in each row
df.apply(num_missing, axis=1).head()
```

Python 3 and Map

Note: Similar to zip, map can return an object (instead of a value) depending on how it's configured. For both zip and map, you can use list() to get the values.

```
def Example1(stuff):
  return stuff + ' THINGS'

#Try this without list, obverse the NewColumn values which are returned as objects
df2 = df.copy()
df2['NewColumn'] = map(Example1, df2['name'])
df2.head()
```

```
#Now try with a list, problem solved when using this syntax
df2 = df.copy()
df2['NewColumn'] = list(map(Example1, df2['name']))
df2.head()
```

Advanced Multi-Column Functions

Note: This is a technique to modify or create multiple columns based on a function that outputs multiple values in a tuple. I've written this to work directly with the sample DataFrame imported at the beginning of this resource guide.

Example2 outputs a tuple of (x, y, z) which we unpack from map using * and then zip inline.

```
def Example2(one, two, three):
  x = str(one)+' Text '+str(two)+' Text '+str(three)
  y = sum([one, two, three])
  z = 'Poptarts'
  return x, y, z

df2 = df.copy()

df2['StrColumn'], df2['SumColumn'], df2['PopColumn'] = zip(*map(
  Example2, df2['mpg'], df2['cyl'], df2['hp']))

df2.head()
```

Conditionally Updating Values

Use `.loc` to update values where a certain condition has been met. This is analogous to `SET ... WHERE ...` syntax in SQL.

```
df2 = df.copy()
df2['efficiency'] = ""
# in SQL, "UPDATE <tablename> SET efficiency = 'poor' WHERE mpg < 10"
df2.loc[(df2.mpg < 10), 'efficiency'] = "poor"
df2.loc[(df2.mpg >= 10) & (df2.mpg < 30), 'efficiency'] = "medium"
df2.loc[(df2.mpg >= 30), 'efficiency'] = "high"
df2.tail()
```

GroupBy and Aggregate

Doc ID: 6689695

Pandas makes it pretty simply to group your dataframe on a value or values, and then aggregate the other results. It's a little less flexible than SQL in some ways, but still pretty powerful. There's a lot you can do in Pandas with GroupBy objects, so definitely check the documentation.

```
# setting as_index to False will keep the grouped values as
# regular columns values rather than indices
grouped_df = df.groupby(by=['cyl'])

# use .agg to aggregate the values and run specified functions
# note that we can't create new columns here
aggregated = grouped_df.agg({
  'mpg': np.mean,
  'displ': np.mean,
  'hp': np.mean,
  'yr': np.max,
  'accel': 'mean'
})
aggregated.head()
```

COMP3321 - A bit about geos

0

Created almost 3 years ago by [redacted] in COMP 3321

3 249 72

fcx91 bokeh geopandas geopy

(U) This notebook gives an overview of some basic geolocation functionality within Python.

Recommendations

(U) A bit about geos

(U) This notebook touches some of the random Python geolocation functionality.

```
# Run this if on LABBENCH
# NOTE: geopandas REQUIRES  running 'apk add geos gdal-dev'
# from a terminal window.
import ipydeps

packages = ['geopy','geopandas','bokeh']
for i in packages:
  ipydeps.pip(i)
```

(U) Measuring Distance

(U) Geopy can calculate geodesic distance between two points using the Vincenty distance or great-circle distance formulas, with a default of Vincenty available as the class `geopy.distance.distance`, and the computed distance available as attributes (e.g., `miles`, `meters`, etc.).

```
from geopy.distance import vincenty
fort_meade_md = (39.10211545,-76.7460704220387)
aurora_co = (39.729432,-104.8319196)
print(vincenty(fort_meade_md, aurora_co).miles, "Miles")
```

```
from geopy.distance import great_circle
harrogate_uk = (53.9921491,-1.5391039)
aurora_co = (39.729432,-104.8319196)
print(great_circle(harrogate_uk, aurora_co).kilometers, "Kilometers")
```

(U) Getting crazy with Bokeh and Maps!

(U) This information comes from this great notebook.

(U) We can add map tiles to Bokeh plots to better show geolocation information! We will use some generic lat/lon data, found here

```
import pandas as pd
us_cities = pd.DataFrame().from_csv('us_cities.csv',index_col=None)
us_cities.head()
```

(U) Define the WMTS Tile Source

(U//~~FOUO~~) Adding the tile source is as easy as defining the WMTS Tile Source, and adding the tile to the the map. Note: you need your Intelink VPN spun up to connect to this server.

```
from bokeh.models import WMTSTileSource, TMSTileSource
```

(U) You also need to convert the lat and lon to plot correctly on the mercator projection map:

```
import math

###METHODS FOR LAT/LONG TICK FORMATTING
def projDegToRad(deg):
    return (deg / 180.0 * math.pi)

def lat_lon_convert(n,lat_or_lon,isDeg=True):
    sm_a = 6378137.0
    sm_b = 6356752.314

    n = float(n)
    lon0 = 0.0
    if isDeg:
        n = projDegToRad(n)

    if lat_or_lon == 'latitude':
        return sm_a * math.log((math.sin(n)+1.0)/math.cos(n))

    elif lat_or_lon == 'longitude':
        return sm_a*(n-lon0)

us_cities['merc_x'] = list(map(lambda x: lat_lon_convert(x,'longitude'),us_cities.lng))
us_cities['merc_y'] = list(map(lambda x: lat_lon_convert(x,'latitude'),us_cities.lat))
```

(U) Finally, plot the data!

```
from bokeh.plotting import output_notebook, show
from bokeh.charts import Scatter
from bokeh.resources import INLINE
output_notebook(resources=INLINE)
p = Scatter(us_cities, x='merc_x', y='merc_y', title="Positions of US Cities")
p.add_tile(NGA_MAP) #vpn is necessary
show(p)
```

(U) GeoPandas

(U) `GeoPandas` is a project to add support for geographic data to `pandas` objects. It currently implements `GeoSeries` and `GeoDataFrame` types which are subclasses of `pandas.Series` and `pandas.DataFrame` respectively. GeoPandas objects can act on `shapely` geometry objects and perform geometric operations.

Doc ID: 6689695

(U) GeoPandas geometry operations are cartesian. The coordinate reference system (crs) can be stored as an attribute on an object, and is automatically set when loading from a file. Objects may be transformed to new coordinate systems with the `to_crs()` method. There is currently no enforcement of like coordinates for operations, but that may change in the future.

```
from geopandas import GeoDataFrame
from matplotlib import pyplot as plt
from shapely.geometry.polygon import Polygon
from descartes import PolygonPatch
poly = Polygon([(1,1),(1,2),(1.5,3),(7,7),(5,4),(2,3)])
BLUE = '#6699cc'
fig = plt.figure()
ax = fig.gca()
ax.add_patch(PolygonPatch(poly, fc=BLUE, alpha=0.5, zorder=2))
ax.axis('scaled')
plt.show()
```

Module: My First Web Application

0

Updated 11 months ago by [redacted] in COMP 3321

3 1 247 62

fcs6 python

(U) Module: My First Web Application

Recommendations

(U) A Word On Decorators

(U) We've learned that functions can accept functions as parameters, and functions can return functions. Python has a bit of special notation, called decorators, that handles the situation where you want to add extra functionality to many different functions. It's more likely that you will need to *use* and *understand* decorators than it is that you would need to *write* one, but you should still understand the basics of what's going on.

(U) Suppose there are a several functions, all returning strings, that you want to "sign," i.e. append your name to.

```
def doubler(to_print):
    return to_print*2
```

```
def tripler(to_print):
    return to_print*3
```

```
doubler("Hello!\n")
```

(U) We an define a function that accepts this function and wraps it up with the functionality that we want:

```
def signer(f):
    def wrapper(to_print):
        return f(to_print) + '\n--Mark'
    return wrapper
```

(U) To reiterate: the **argument** to `signer` is a function, and the **return value** of `signer` is also a function. It is a function that takes the *same arguments* as the argument `f` passed into `signer`, and inside `wrapper`, `f` is called on those arguments. That is why something like this works:

```
signed_doubler = signer(doubler)
```

```
signed_tripler = signer(tripler)
```

```
signed_doubler("Hello!\n")
```

```
signed_tripler("Hello!\n")
```

(U) If we are willing to replace the original function entirely, we can use the decorator syntax:

```
@signer
def quadrupler(to_print):
    return to_print*4
```

```
quadrupler("Hello!\n")
```

(U) Things get more complicated from there; in particular

- A function can have attributes. Therefore, a decorator can instrument a function by attaching local variables and doing something to them.
- A decorator that takes arguments must generate and return a valid decorator-style function using those arguments.
- A decorator may wrap functions of unknown signature by using `(*args, **kwargs)`.

(U) All of this is useful when working with a complicated, multi-layer system, where much of the work would appear to be repetitive boiler plate. It's best to make the "business logic" (i.e. whatever makes this program unique) as clean and concise as possible by separating it from the scaffolding.

(U) The Flask Framework

(U) **Flask** is a "micro-framework", which means that it handles mostly just the web serving–receiving and parsing HTTP requests, and sending back properly formatted responses. In contrast, a macro-framework, e.g. **Django**, includes its own ORM for database operations, has an integrated framework for users and authentication, and an easy-to-configure administrative backend. Because it offers so much, it takes a long time to get started with Django.

P.L. 86-36

```
(VENV)[ ~]$ pip install flask
(VENV)[ ~]$ python
```

```
import ipydeps
ipydeps.pip('flask')
```

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
def hello():
    return "Hello World"
```

```
app.run(host='0.0.0.0',port=8999) # open ports:8000-9000
```

(U) Press `<Ctrl-c>` to stop your app.

(U) View Functions

(U) A **view function** is anything for which a route has been determined, using the `@app.route` decorator. It can return a variety of types–we've already seen a string, but it can also return a rendered template or a `flask.Response`, which we might use if we want to set custom headers.

```
from flask import request, make_response, redirect, url_for
```

```
fruit = {'apple':'red','banana':'yellow',
         'cranberry':'crimson','date':'brown'}
```

```
@app.route('/fruits/')
def fruit_list():
    fruit_str = "<br />".join(["A {} is {}".format(*i) for i in fruit.items()])
    form_str = """<br>Add something:
                  <form method="post">
                  <input type="text" name="fruit_name"></input>
                  <input type="text" name="fruit_color"></input>
                """
    header_str = """<html><head><title>TEST</title></head><body>"""
    footer_str = """</body></html>"""
    return header_str + fruit_str + form_str + footer_str
```

```
@app.route('/fruits/<name>/')
def single_fruit(name):
    if name in fruit.keys():
        return "A {} is {}".format(name,fruit[name])
    else:
        return make_response("ERROR: FRUIT NOT FOUND",404)
```

```
@app.route('/fruits/',methods=['POST',])
def add_fruit():
    print(request.form)
    print(request.data)
    fruit[request.form['fruit_name']] = request.form['fruit_color']
    return redirect(url_for('fruit_list'))
```

```
app.run(host='0.0.0.0',port=8999)
```

(U) Templates

(U) Flask view functions should probably return HTML, JSON, or some other structured data format. As a general rule, it's a bad idea to build these responses as strings, which is what we've done in the simple example. Flask provides the Jinja2 template engine, which allows you to store the core of the responses in separate files and render content dynamically when the view is called. Another nice feature of Jinja2 templates is inheritance, which can help you create and maintain a consistent look and feel across a Flask website.

(U) For a simple Flask app, templates should be located in a directory called `templates` alongside the application module. When operating interactively, the templates folder must be defined explicitly:

```
import os
```

```
templates = os.path.join(os.getcwd(),'templates')
```

```
app = Flask(__name__,template_folder=templates)
```

(U) A Jinja2 template can have variables, filters, blocks, macros, and expressions, but cannot usually evaluate arbitrary code, so it isn't a full-fledged programming language. It is expected that only small parts of the template will be rendered with dynamic content, so there is a custom syntax optimized for this mode of creation. Variables are surrounded with double curly braces: `{{ '{{ variable }}' }}`, and are typically injected as keyword arguments when the `render_template` function is called. Attributes of objects and dictionaries can be accessed in the normal way. Blocks, conditionals, for loops, and other expressions are enclosed in a curly brace and percent sign, e.g `{{ '{% if condition %}...{% else %}...{% endif %}' }}`.

Doc ID: 6689695

```
from flask import render_template
```

```
@app.route('/fruits/')
def fruit_list():
    return render_template('fruit_list.html',fruits=fruit)
```

```
@app.route('/fruits/<name>/')
def single_fruit(name):
    if name in fruit.keys():
        return render_template('single_fruit.html',name=name,color=fruit[name])
    else:
        return make_response("ERROR: FRUIT NOT FOUND",404)
```

```
app.run(host='0.0.0.0',port=8999)
```

(U) In this example, we also see how template inheritance can be used to isolate common elements and boilerplate. The templates used are:

- base.html

```
<html>
  <head>
    <title>Fruit Stand</title>
  </head>
  <body>
    {% block body %}
    {% endblock %}
  </body>
</html>
```

- fruit_list.html

```
{% extends "base.html" %}
{% block body %}
<h1>Fruit Stand</h1>
<p>Available fruit:</p>
<ul>
{% for fruit in fruits.items() %}
<li><a href="/fruits/{{ fruit[0] }}/">{{ fruit[0] }}</a> ({{ fruit[1]     }})</li>
{% endfor %}
</ul>
<p>Add a new fruit:</p>
<form method="post">
  <label for="fruit_name">Name</label> <input type="text" name="fruit_name"></input><br>
  <label for="fruit_color">Color</label> <input type="text" name="fruit_color"></input><br>
  <input type="submit" value="submit" />
</form>

{% endblock %}
```

- `single_fruit.html`

```
{% extends "base.html" %}
{% block body %}
    <p>A {{ name }} is {{ color }}.</p>
{% endblock %}
```

(U) Moving To Production

(U) In the real world, `app.run()` probably won't get the job done, because

- It isn't designed for performance,
- It doesn't handle HTTPS or PKI gracefully,
- It doesn't handle more than one request at a time.

(U) However, Flask makes our `app` conform to the WSGI standard, so it interoperates very easily with Python web-server containers, including **uWSGI** and **gunicorn**.

(U) A high performance stack for a production web application is:

- **nginx**: a fast, lightweight front-end server proxy that can receive HTTPS requests and pass them to **gunicorn**, taking care to add PKI authentication headers.
- **supervisord**: Process manager to make sure your app never dies.

- **gunicorn**: serves your flask app on a (closed, internal) port
- **flask**: framework in which you write an app
- **your app**: takes care of all the business logic
- **database**: SQLite if you don't need much, MySQL or Postgres if necessary.

(U) Another option is to use **Apache** with **mod_wsgi** instead of **nginx**, **supervisord**, and **gunicorn**.