Basic Jupyter Notebook Tutorial

This lab has been iteratively developed for EE 126 at UC Berkeley by Rishi Sharma, Sahaana Suri, Kangwook Lee, Kabir Chandrasekher, Max Kanwal, Tony Duan, David Marn, Ashvin Nair, Tavor Baharav, Sinho Chewi, Andrew Liu, Kamil Nar, David Wang, Avishek Ghosh, Chen Meng, Alvin Kao, Ray Ramamurti, Nikunj Jain, Kevin Lu and Professors Kannan Ramchandran, Abhay Parekh, and Jean Walrand.

Modified from Berkeley Python Bootcamp 2013 https://github.com/profjsb/python-bootcamp (https://github.com/profjsb/python-bootcamp)

and Python for Signal Processing http://link.springer.com/book/10.1007%2F978-3-319-01342-8 (http://link.springer.com/book/10.1007%2F978-3-319-01342-8)

and EE 123 iPython Tutorial http://inst.eecs.berkeley.edu/~ee123/sp14/lab/python_tutorial.ipynb (http://inst.eecs.berkeley.edu/~ee123/sp14/lab/python_tutorial.ipynb).

This lab serves as a quick overview of Jupyter Notebooks, numpy, matplotlib, and simulations in Python, which we will use throughout the course. If you are unfamiliar with these, you should carefully read through all of the examples. You are only required to answer the two questions at the bottom for credit.

General Jupyter Notebook Usage Instructions (Overview)

- Start by clicking `Help >> User Interface Tour` to get yourself familiar with the Jupyter Notebook environment.
- Click the `Play` button to run and advance a cell. The short-cut for it is `Shift-Enter`.
- To add a new cell, either select `"Insert >> Insert New Cell Below"` or click the `Plus` button.
- You can change the cell mode from code to text in the pulldown menu. Use `Markdown` for writing text.
- You can change the text in `Markdown` cells by double-clicking it. The short-cut for this is `enter`.
- To save your notebook, either select `"File >> Save and Checkpoint"` or hit `Command-s` for Mac and `Ctrl-s` for Windows.
- To undo edits within a cell, hit `Command-z` for Mac and `Ctrl-z` for Windows.
- `Help >> Keyboard Shortcuts` has a list of all useful keyboard shortcuts.
- The `Help` menu also has links to many reference docs you may find useful this semester (e.g. Markdown, Python, NumPy, Matplotlib, SciPy).

Tab Completion

One useful feature of iPython is tab completion:

In [1]:
```
x = 1
y = 2
x_plus_y = x + y

# Type `x_` then hit TAB to auto-complete the variable and then press Sh
# Enter to run the cell.
print(x_plus_y)
```

```
3
```

Help

Another useful feature is the help command. Type any function followed by `?` and run the cell to return a help window. Hit the `x` button to close it.

In [2]:
```
abs?
```

```
Signature: abs(x, /)
Docstring: Return the absolute value of the argument.
Type:      builtin_function_or_method
```

Floats and Integers

Doing math in Python is easy, but note that there are `int` and `float` types in Python. In Python 3, integer division returns the same results as floating point division.

In [3]:
```
59 / 87
```

Out[3]:
```
0.6781609195402298
```

In [4]:
```
59 / 87.0
```

Out[4]:
```
0.6781609195402298
```

Strings

- Double quotes and single quotes are the same thing.
- `'+'` concatenates strings.

In [5]:
```
# This is a comment.
"Hi " + 'Bye'
```

Out[5]:
```
'Hi Bye'
```

Printing

Here are some fancy ways of printing:

In [6]:
```
speed_of_light = 299792458
speed_of_sound = 343
```

In [7]:
```
print("Light travels at %d meters per second. This can be formatted as:
      % (speed_of_light, speed_of_light))
```

```
Light travels at 299792458 meters per second. This can be formatted a
s: 3.00E+08 meters per second.
```

In [8]:
```
print ("The speed of sound is {0} meters per second. That is {1}% the sp
    .format(speed_of_sound, speed_of_sound / speed_of_light * 100))
```

```
The speed of sound is 343 meters per second. That is 0.000114412484652
96614% the speed of light.
```

In [9]:
```
print("Good Luck! Prepare to work hard and learn a lot of cool stuff!")
```

```
Good Luck! Prepare to work hard and learn a lot of cool stuff!
```

Lists

A list is a mutable array of data, i.e. it can constantly be modified. See http://stackoverflow.com/questions/8056130/immutable-vs-mutable-types-python (http://stackoverflow.com/questions/8056130/immutable-vs-mutable-types-python) for more info. If you are not careful, using mutable data structures can lead to bugs in code that passes common data to many different functions.

Important functions:

- Created a list by using square brackets `[ ]`.
- `'+'` appends lists.
- `len(x)` gets the length of list `x`.

In [10]:
```
x = [1, 2, "asdf"] + [4, 5, 6]

print(x)
```

```
[1, 2, 'asdf', 4, 5, 6]
```

In [11]:
```
print(len(x))
```

```
6
```

Tuples

A tuple is an immutable list. They can be created using round brackets ( ).

They are usually used as inputs and outputs to functions.

In [12]:
```
t = (1, 2, "asdf") + (3, 4, 5)
print(t)
```

```
(1, 2, 'asdf', 3, 4, 5)
```

In [13]:
```
# cannot do assignment
t[0] = 10

# errors in Jupyter Notebook appear inline
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-13-134c6ebd92fa> in <module>
      1 # cannot do assignment
----> 2 t[0] = 10
      3 
      4 # errors in Jupyter Notebook appear inline

TypeError: 'tuple' object does not support item assignment
```

Arrays (NumPy)

A NumPy array is like a list with multidimensional support and more functions. We will be using it a lot.

Arithmetic operations on NumPy arrays correspond to elementwise operations.

Important functions:

- `.shape` returns the dimensions of the array.
- `.ndim` returns the number of dimensions.
- `.size` returns the number of entries in the array.
- `len()` returns the first dimension.

To use functions in NumPy, we have to import NumPy to our workspace. This is done by the command `import numpy`. By convention, we rename `numpy` as `np` for convenience.

In [14]:
```
# by convention, import numpy as np
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])

print(x)
```

```
[[1 2 3]
 [4 5 6]]
```

In [15]:
```
print("Number of Dimensions:", x.ndim)
```

```
Number of Dimensions: 2
```

In [16]:
```
print("Dimensions:", x.shape)
```

```
Dimensions: (2, 3)
```

In [17]:
```
print("Size:", x.size)
```

```
Size: 6
```

In [18]:
```
print("Length:", len(x))
```

```
Length: 2
```

In [19]:
```
a = np.array([1, 2, 3])

print("a = ", a)

# elementwise arithmetic
print("a * a = ", a * a)
```

```
a =  [1 2 3]
a * a =  [1 4 9]
```

In [20]:
```
b = np.array(np.ones((3, 3))) * 2
print("b =\n", b)
c = np.array(np.ones((3, 3)))
print("c =\n", c)
```

```
b =
 [[2. 2. 2.]
 [2. 2. 2.]
 [2. 2. 2.]]
c =
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Multiply elementwise:

In [21]:
```
print("b * c =\n", b * c)
```

```
b * c =
 [[2. 2. 2.]
 [2. 2. 2.]
 [2. 2. 2.]]
```

Now multiply as matrices (not arrays):

In [22]:
```
print("b * c =\n", np.dot(b, c))
```

```
b * c =
 [[6. 6. 6.]
 [6. 6. 6.]
 [6. 6. 6.]]
```

With Python3, you can also use the "@" operator for the dot product

In [23]:
```
print("b * c =\n", b @ c)
```

```
b * c =
 [[6. 6. 6.]
 [6. 6. 6.]
 [6. 6. 6.]]
```

Alternatively, we can just convert to or create a matrix instead of an array and then use normal multiplication:

In [24]:
```
print("b * c =\n", np.matrix(b) * np.matrix(c))

d = np.matrix([[1, 1j, 0], [1, 2, 3]])
e = np.matrix([[1], [1j], [0]])

print("d * e =\n", d * e)
```

```
b * c =
 [[6. 6. 6.]
 [6. 6. 6.]
 [6. 6. 6.]]
d * e =
 [[0.+0.j]
 [1.+2.j]]
```

Slicing for NumPy Arrays

NumPy uses pass-by-reference semantics so it creates views into the existing array, without implicit copying. This is particularly helpful with very large arrays because copying can be slow.

In [25]:
```
x = np.array([1, 2, 3, 4, 5, 6])
print(x)
```

```
[1 2 3 4 5 6]
```

We slice an array from `a` to `b - 1` with `[a:b]` .

In [26]:
```
y = x[0:4]
print(y)
```

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

Since slicing does not copy the array, changing `y` changes `x`:

In [27]:
```
y[0] = 7
print(x)
print(y)
```

```
[7 2 3 4 5 6]
[7 2 3 4]
```

To actually copy `x`, we should use `.copy`:

In [28]:
```
x = np.array([1, 2, 3, 4, 5, 6])
y = x.copy()
y[0] = 7
print(x)
print(y)
```

```
[1 2 3 4 5 6]
[7 2 3 4 5 6]
```

Plotting

In this class we will use `matplotlib.pyplot` to plot signals and images.

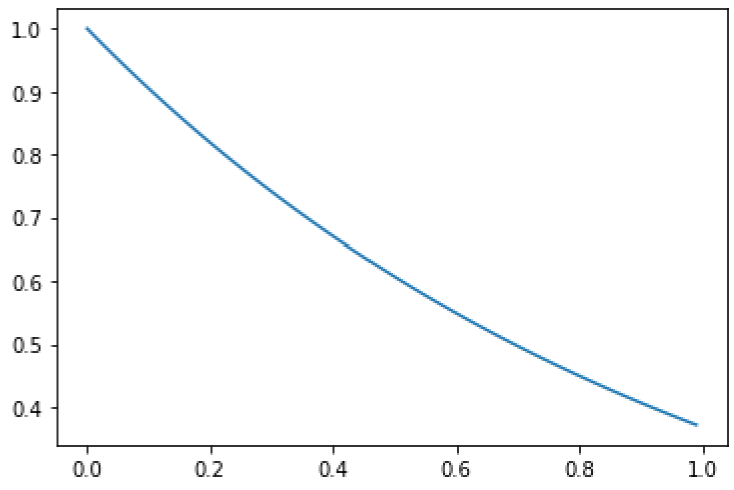
To begin with, we import `matplotlib.pyplot` as `plt` (again for convenience).

In [29]:
```
import numpy as np
# by convention, we import pyplot as plt
import matplotlib.pyplot as plt
# import r_ function from numpy
from numpy import r_


# if you don't specify a number before the colon, the starting index de
# to 0
x = r_[:1:0.01]
a = np.exp(-x)
b = np.sin(x * 10.0) / 4.0 + 0.5

# plot in browser instead of opening new windows
%matplotlib inline
```

`plt.plot(x, a)` plots `a` against `x`.

In [30]:
```
plt.figure()
plt.plot(x, a)
```

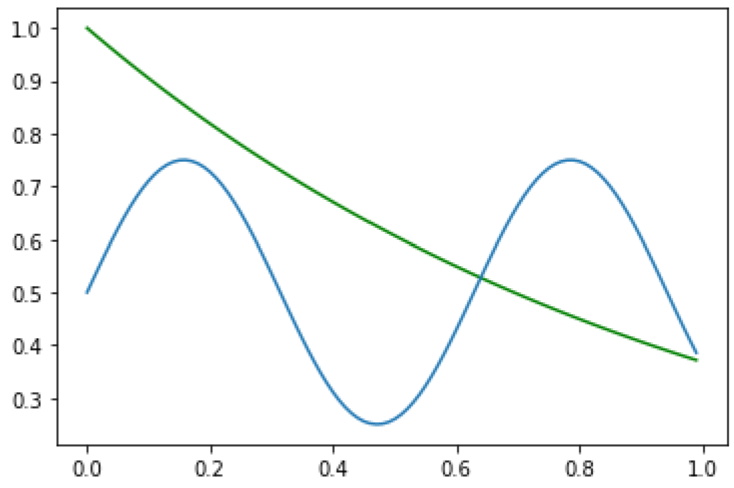
Out[30]:
```
[<matplotlib.lines.Line2D at 0x115232e50>]
```



Once you started a figure, you can keep plotting to the same figure.

In [31]:
```
plt.figure()
plt.plot(x, a, "green")
plt.plot(x, b)
```

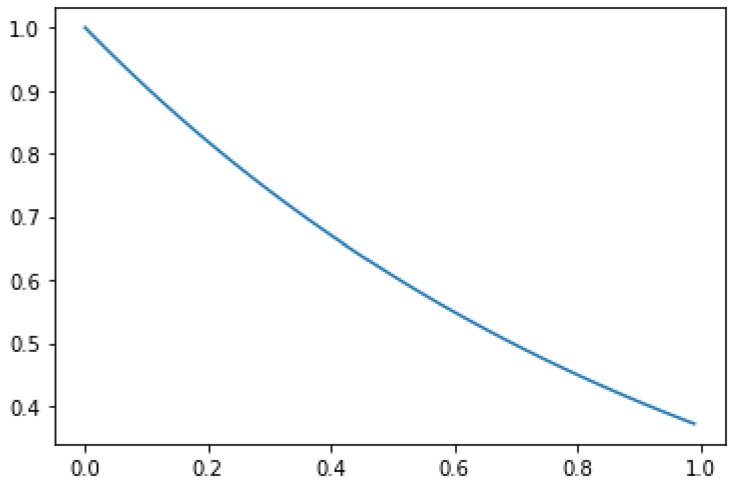
Out[31]:
```
[<matplotlib.lines.Line2D at 0x114b48a10>]
```

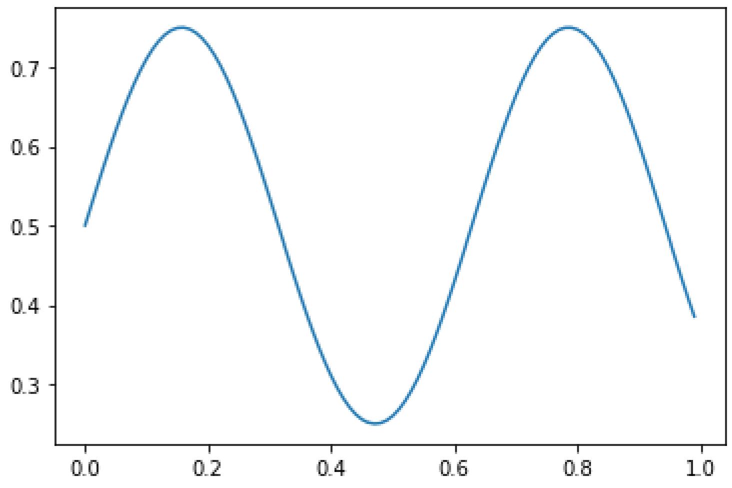


To plot different plots, you can create a second figure.

In [32]:
```
plt.figure()
plt.plot(x, a)
plt.figure()
plt.plot(x, b)
```

Out[32]:
```
[<matplotlib.lines.Line2D at 0x11573b610>]
```

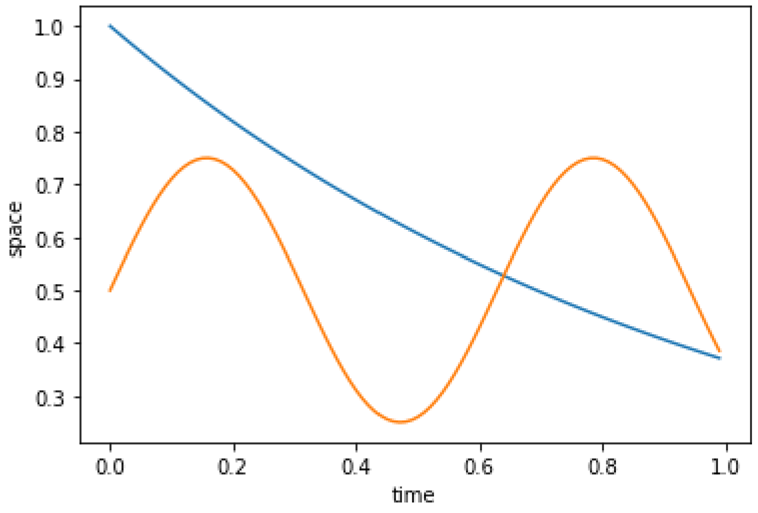




To label the axes, use `plt.xlabel()` and `plt.ylabel()` .

In [33]:
```
plt.figure()
plt.plot(x, a)
plt.plot(x, b)

plt.xlabel("time")
plt.ylabel("space")
```

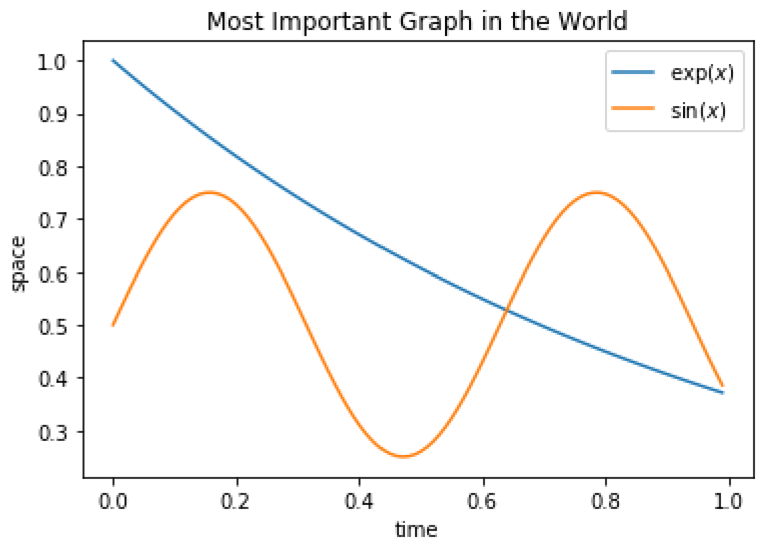
Out[33]:
```
Text(0, 0.5, 'space')
```



You can also add title and legends using `plt.title()` and `plt.legend()` .

In [34]:
```
plt.figure()
plt.plot(x, a)
plt.plot(x, b)
plt.xlabel("time")
plt.ylabel("space")

plt.title("Most Important Graph in the World")

plt.legend(("$\exp(x)$", "$\sin(x)$"))
```

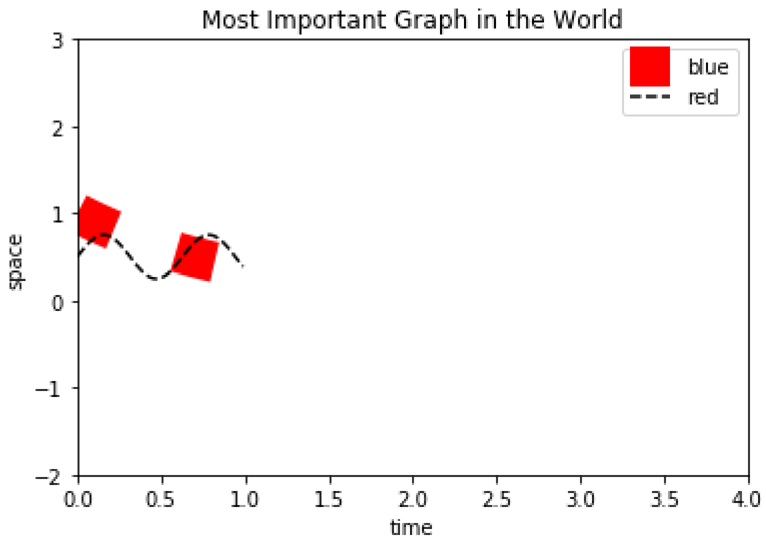
Out[34]:
```
<matplotlib.legend.Legend at 0x11584c490>
```



There are many options you can specify in `plot()` , such as color and linewidth. You can also change the axis using `plt.axis` .

In [35]:
```
plt.figure()
plt.plot(x, a, ":r", linewidth=20)
plt.plot(x, b , "--k")
plt.xlabel("time")
plt.ylabel("space")

plt.title("Most Important Graph in the World")

plt.legend(("blue", "red"))

plt.axis([0, 4, -2, 3])
```

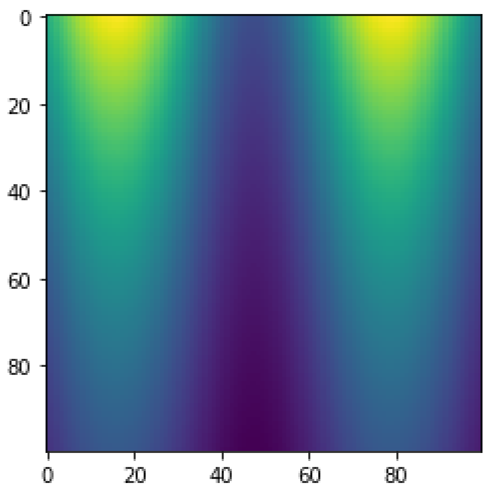
Out[35]:
```
[0, 4, -2, 3]
```



There are many other plotting functions. For example, we will use `plt.imshow()` for showing images and `plt.stem()` for plotting discretized signals.

In [36]:
```
# image
plt.figure()

# plotting the outer product of a and b
data = np.outer(a, b)

plt.imshow(data)
```

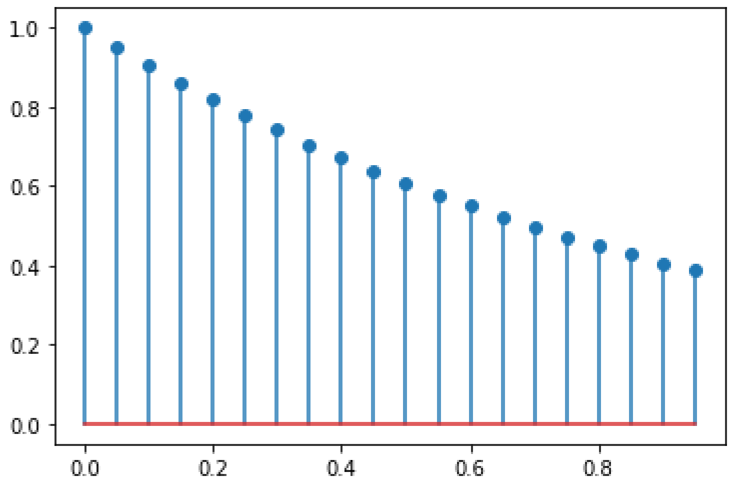
Out[36]:
```
<matplotlib.image.AxesImage at 0x115c17510>
```



In [37]:
```
# stem plot
plt.figure()
# subsample by 5
plt.stem(x[::5], a[::5])
```

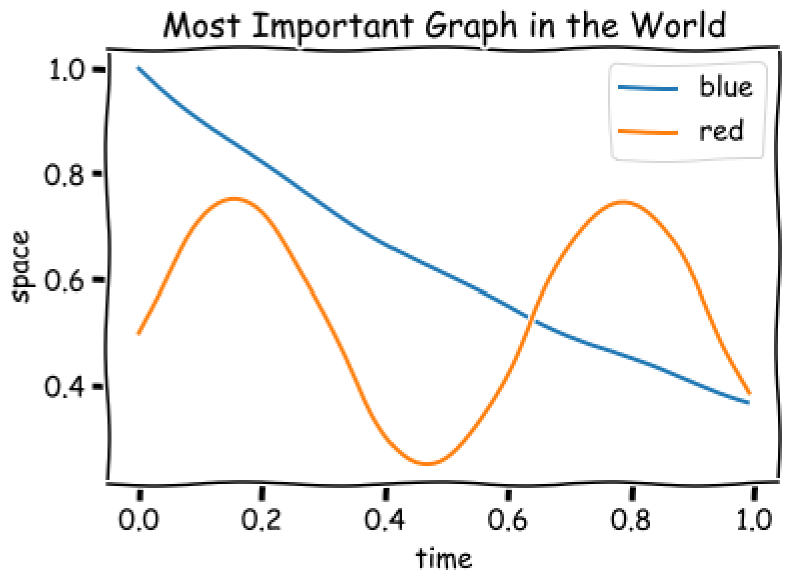
```
/Users/wjgan/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.
py:4: UserWarning: In Matplotlib 3.3 individual lines on a stem plot w
ill be added as a LineCollection instead of individual lines. This sig
nificantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection" ke
yword argument to True.
  after removing the cwd from sys.path.
```

Out[37]:
```
<StemContainer object of 3 artists>
```



In [38]:
```
# xkcd style plots
# Note: Requires matplotlib version 1.3.1 or higher
plt.xkcd()
plt.plot(x, a)
plt.plot(x, b)
plt.xlabel("time")
plt.ylabel("space")

plt.title("Most Important Graph in the World")

plt.legend(("blue", "red"))
```

Out[38]:
```
<matplotlib.legend.Legend at 0x115c9ee50>
```



To turn off xkcd style plotting, restart the kernel or run the command `plt.rcdefaults()`.

Logic

For Loop

Indentation matters in Python. Everything indented belongs to the loop:

In [39]:
```
for i in [4, 6, "asdf", "jkl"]:
    print(i)
```

```
4
6
asdf
jkl
```

In [40]:
```
for i in np.arange(0, 1, 0.1):
    print(i)
```

```
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6000000000000001
0.7000000000000001
0.8
0.9
```

If-Else

Same goes for If-Else:

In [41]:
```
if 1 != 0:
    print("1 != 0")
elif 1 == 0:
    print("1 = 0")
else:
    print("Huh?")
```

```
1 != 0
```

Random Library

The NumPy random library should be your resource for all Monte Carlo simulations which require generating instances of random variables.

The documentation for the library can be found here:
http://docs.scipy.org/doc/numpy/reference/routines.random.html (http://docs.scipy.org/doc/numpy/reference/routines.random.html)

The function `rand` can be used to generates a uniform random number in the range $[0, 1)$.

In [42]:
```
from numpy import random

# random number
print(random.rand())
# random vector
print(random.rand(5))
# random matrix
print(random.rand(3, 3))
```

```
0.30827390446658687
[0.43193358 0.89712897 0.25453753 0.6048372  0.78784836]
[[0.38848219 0.3484001  0.67090501]
 [0.2018519  0.61155685 0.04911921]
 [0.67024566 0.3083388  0.6769348 ]]
```

Let's see how we can use this to generate a fair coin toss (i.e. a discrete $\text{Bernoulli}(1/2)$ random variable).

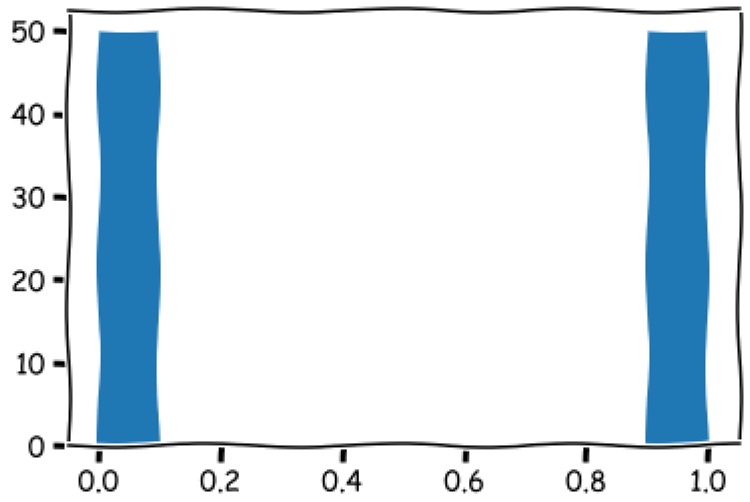
In [43]:
```
# Bernoulli(1/2) random variable
x = round(random.rand())
print(x)
```

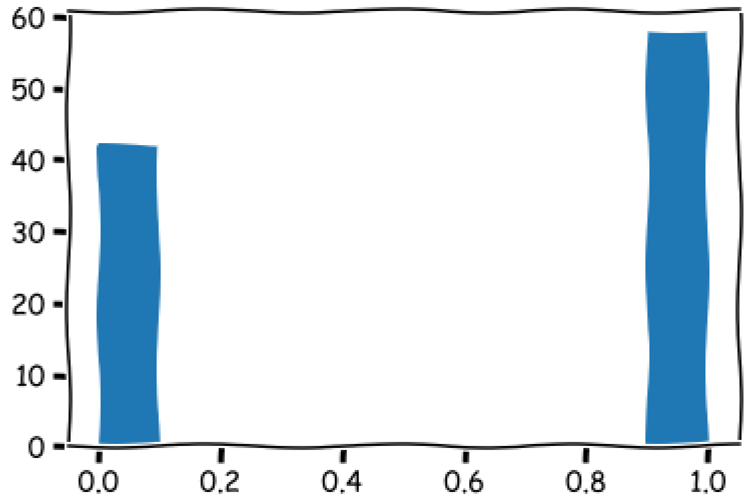
```
0
```

Now let's generate several fair coin tosses and plot a histogram of the results.

```
In [44]: k = 100
         x1 = [round(random.rand()) for _ in range(k)]
         plt.figure()
         plt.hist(x1)

         # we could also use NumPy's round function to elementwise round the vect
         x2 = np.round(random.rand(k))
         plt.figure()
         plt.hist(x2)
```

```
Out[44]: (array([42.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., 58.]),
          array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
          <a list of 10 Patch objects>)
```

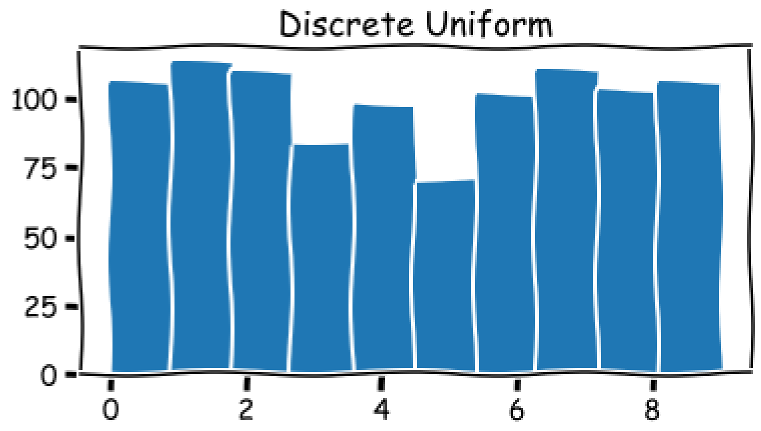


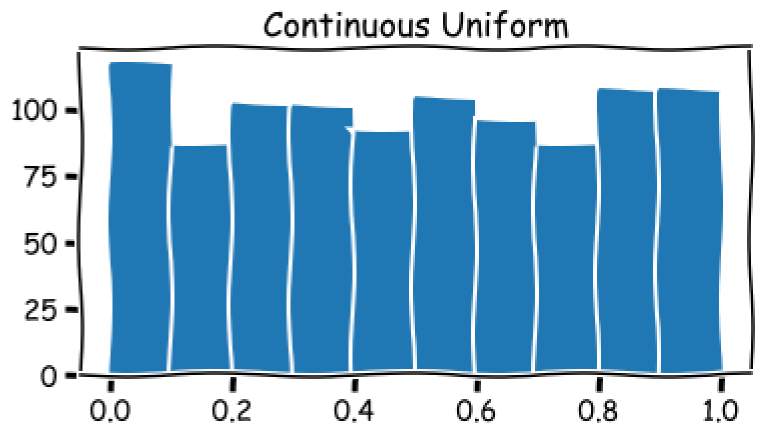


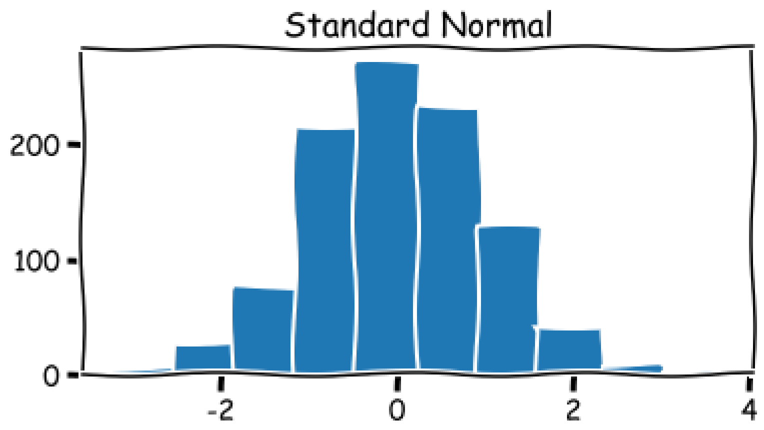
We can do something similar for several other distributions, and allow the histogram to give us a sense of what the distribution looks like. As we increase the number of samples we take from the distribution k, the more and more our histogram looks like the actual distribution.

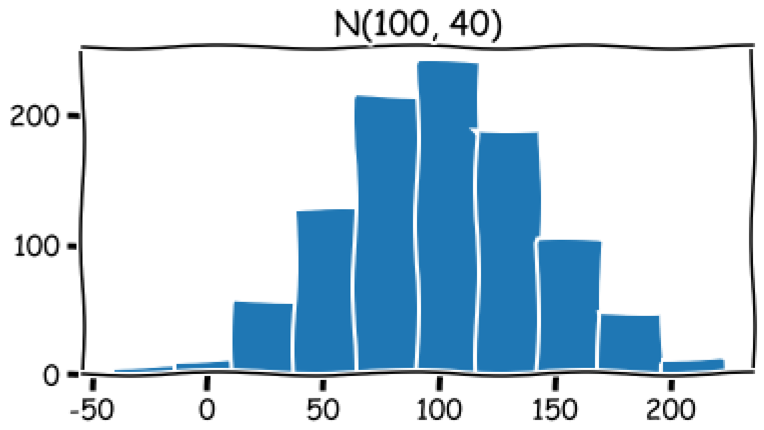
In [45]:
```python
k = 1000

# k discrete uniform random variables between 0 and 9
discrete_uniform = random.randint(0, 10, size=k)
plt.figure(figsize=(6, 3))
plt.hist(discrete_uniform)
plt.title("Discrete Uniform")

continuous_uniform = random.rand(k)
plt.figure(figsize=(6, 3))
plt.hist(continuous_uniform)
plt.title("Continuous Uniform")

# randn generates elements from the standard normal
std_normal = random.randn(k)
plt.figure(figsize=(6, 3))
plt.hist(std_normal)
plt.title("Standard Normal")

# To generate a normal distribution with mean mu and standard deviation
# we must mean shift and scale the variable
mu = 100
sigma = 40
normal_mu_sigma = mu + random.randn(k) * sigma
plt.figure(figsize=(6, 3))
plt.hist(normal_mu_sigma)
plt.title("N({}, {})".format(mu, sigma))
```

Out[45]:
```
Text(0.5, 1.0, 'N(100, 40)')
```









^ We could do this all day with all sorts of distributions. I think you get the point.

Specifying a Discrete Probability Distribution for Monte Carlo Sampling

The following function takes n sample from a discrete probability distribution specified by the two arrays `distribution` and `values` .

As an example, let us suppose a random variable X follows the following distribution:

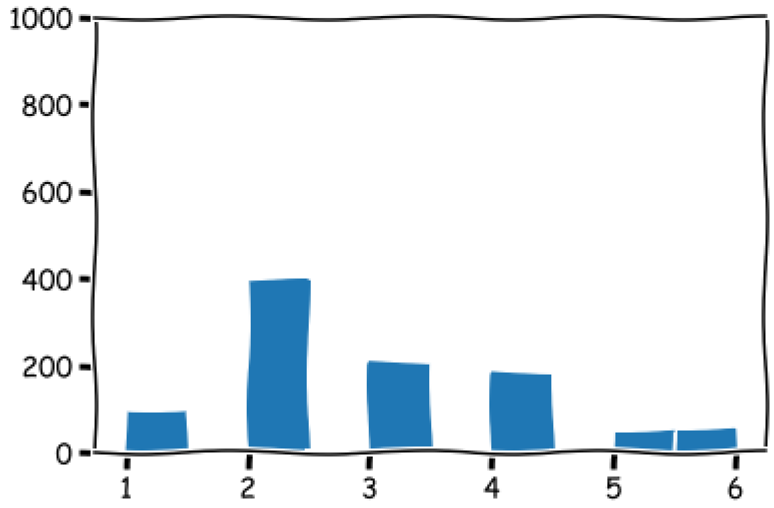
$$X = \begin{cases} 1 \text{ w/ probability } 0.1 \\ 2 \text{ w/ probability } 0.4 \\ 3 \text{ w/ probability } 0.2 \\ 4 \text{ w/ probability } 0.2 \\ 5 \text{ w/ probability } 0.05 \\ 6 \text{ w/ probability } 0.05 \end{cases}$$

Then we would have: `distribution = [0.1, 0.4, 0.2, 0.2, 0.05, 0.05]` and `values = [1, 2, 3, 4, 5, 6]` .

In [46]:
```
def n_sample(distribution, values, n):
    if sum(distribution) != 1:
        distribution = [distribution[i] / sum(distribution) \
            for i in range(len(distribution))]
    rand = [random.rand() for i in range(n)]
    rand.sort()
    samples = []
    sample_pos, dist_pos, cdf = 0, 0, distribution[0]
    while sample_pos < n:
        if rand[sample_pos] < cdf:
            sample_pos += 1
            samples.append(values[dist_pos])
        else:
            dist_pos += 1
            cdf += distribution[dist_pos]
    return samples
```

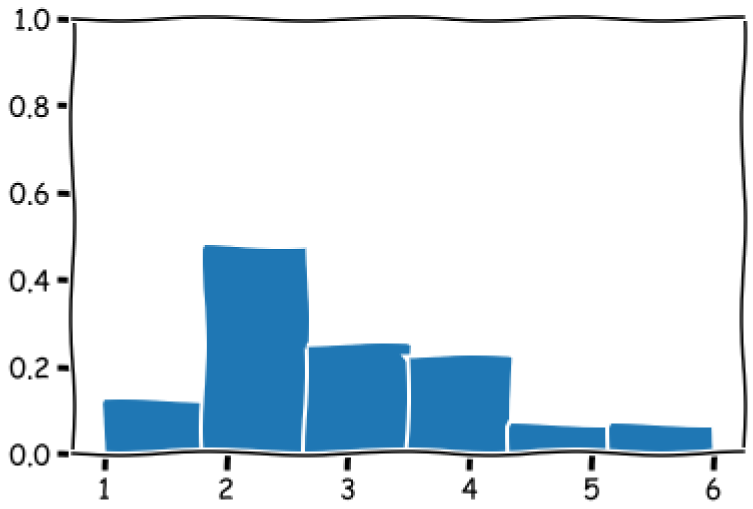
In [47]:
```
# collect k samples from X and plot the histogram
samples_from_x = n_sample(
    [0.1, 0.4, 0.2, 0.2, 0.05, 0.05], [1, 2, 3, 4, 5, 6], k)
plt.hist(samples_from_x)
plt.ylim((0, 1000))
print("Wow, if we normalized the y-axis that would be a PMF. Incredible
print("I should try that.")
```

```
Wow, if we normalized the y-axis that would be a PMF. Incredible!
I should try that.
```



In [49]:
```
# Normalize the samples from X to plot the probability mass function be
plt.hist(samples_from_x, bins=6, density=True)
plt.ylim((0, 1))
```

Out[49]:
```
(0, 1)
```

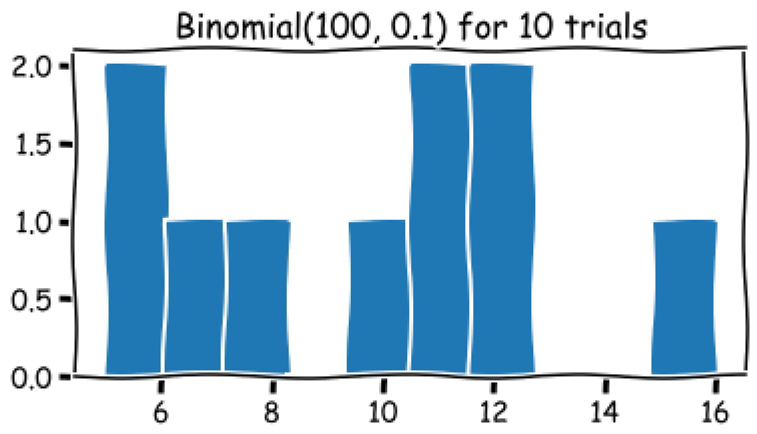


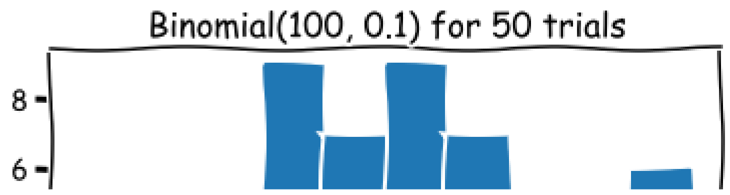
$\mathcal{Q}$uestion 1: Sampling and Plotting a Binomial Random Variable

A binomial random variable $X \sim \text{Binomial}(n, p)$ can be thought of as the number of heads in n coin flips where each flip has probability p of coming up heads. We can equivalently think of it as the sum of n Bernoulli random variables: $X = \sum_{i=1}^{n} X_i$ where $X_i \sim \text{Bernoulli}(p)$.

In this question, you will put your new plotting skills to work and sample the values of a binomial random variable.

In [50]:
```
def plot_binomial(
        n, trials=[10, 50, 100, 1000, 10000], p_values=[0.1, 0.2, 0.5, 0
    """
    On different figures, plot a histogram of the results of the given
    number of trials of the binomial variable with parameters n and p fo
    values in the given list.

    You should generate 20 different plots in total.
    """
    for p in p_values:
        for num_trials in trials:
            binomial = random.binomial(n, p, num_trials)
            plt.figure(figsize=(6, 3))
            plt.hist(binomial)
            plt.title(f"Binomial({n}, {p}) for {num_trials} trials")

# Feel free to play around with other values of n.
plot_binomial(100)
```





Now that you have plotted many values of a few different binomial random variables, do the results coincide with what you expect them to?

Question 2: Monte Carlo Method for Estimating Pi

After going through this tutorial, you might wonder: why should we bother with NumPy at all?

While many of you may not have used NumPy previously, or not see the purpose in learning this library, we strongly urge you to force yourself to use NumPy as much as possible while doing virtual labs.

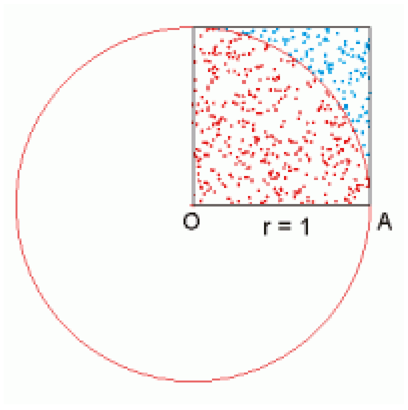
NumPy (and using matrix operations rather than loops) is your friend when it comes to efficiently dealing with lots of data or doing elaborate simulations. If you find yourself using many loops or list comprehensions to process data, think about using NumPy. Furthermore, NumPy is widely used in industry and academic research, and so it will benefit you greatly to become comfortable with it this semester!

Let's work through an example to see the usefulness of NumPy.

Estimate the value of π using a Monte Carlo method.

Monte Carlo methods are algorithms that use probability and randomness to solve problems that would be difficult otherwise. Here, we will use a Monte Carlo simulation to estimate the value of π.

Suppose we have a one meter by one meter square dartboard with a quarter circle inscribed in it, as shown below:



If we throw darts at the dartboard such that they are equally likely to land anywhere in the square, then as we throw more and more darts, the fraction of them that will land in the quarter circle will approach

$$\frac{\text{Area of quarter circle}}{\text{Area of square}} = \frac{\pi}{4}$$

Therefore, after simulating this process, we can take the fraction of darts that landed within the circle, and multiply this value by 4 to use as our estimate for π.

A version of the simulation that does not use NumPy at all is given. Your job is to re-implement the simulation using NumPy to speed it up. For reference, the staff solution is > 10x better.

In [1]:
```
import time
import numpy as np
import random

def monte_carlo_pi(num_points):
    # num_points is the number of random points to choose
    count = 0
    for _ in range(num_points):
        randx, randy = random.random(), random.random()
        if (randx ** 2 + randy ** 2 < 1):
            count += 1
    estimate = (count * 4.0 / num_points)
    return estimate


def monte_carlo_pi_numpy(num_points):
    # num_points is the number of random points to choose

    # Your beautiful code here... #
    x, y = np.random.rand(num_points), np.random.rand(num_points)
    dists = np.add(np.square(x), np.square(y))
    in_circle = np.less_equal(dists, 1)
    return np.count_nonzero(in_circle) / num_points * 4
```

To see the effectiveness of using NumPy, let's run both implementations of the simulation, with and without NumPy, and compare their speeds.

In [2]:
```
num_points = 10000000 # number of random points to choose
```

In [3]:
```
start = time.perf_counter()
print("Estimate of Pi:", monte_carlo_pi(num_points))
print ("Actual value of Pi:", np.pi)
end = time.perf_counter()
total1 = end - start
```

```
Estimate of Pi: 3.141812
Actual value of Pi: 3.141592653589793
```

In [4]:
```
start = time.perf_counter()
print("Estimate of Pi:", monte_carlo_pi_numpy(num_points))
print ("Actual value of Pi:", np.pi)
end = time.perf_counter()
total2 = end - start
```

```
Estimate of Pi: 3.141036
Actual value of Pi: 3.141592653589793
```

In [5]:
```
print("w/o NumPy:\t %f s\nw/ NumPy:\t %f s" %(total1, total2))
print("Total Speedup: " + str(total1 / total2) + "x")
```

```
w/o NumPy:       3.121263 s
w/ NumPy:        0.167935 s
Total Speedup: 18.586183257367896x
```

In [ ]:

```
```