PSYC 027: Scientific Computing for Psychology

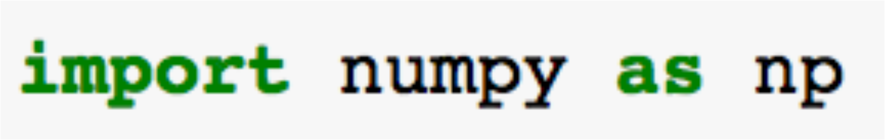
8 October 2019

Professor Youssef Ezzyat
McCabe Library 306
T/Th 9:55-11:10

Numpy and Scipy

- Numpy = numerical python
  - Core library for mathematical/scientific computing in Python

- Scipy = scientific Python
  - Extended library for mathematical/scientific computing
  - Relies on Numpy



```
import numpy as np
```

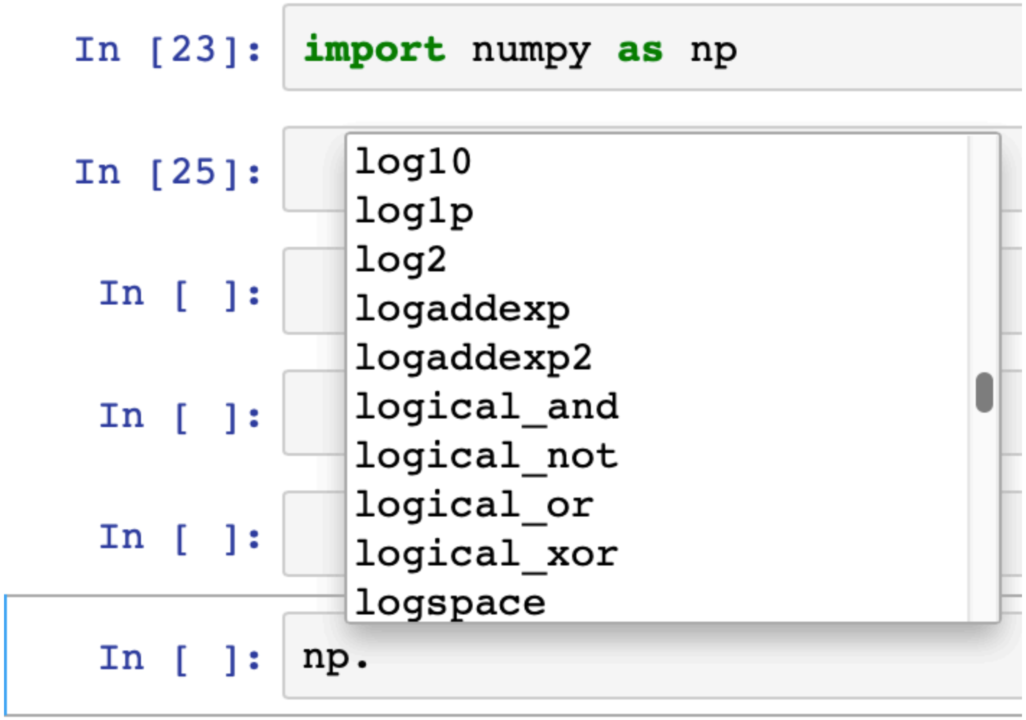
Why Numpy?

- Provides an easy to use data structure (the Numpy array) that can represent multi-dimensional data

- Optimizes calculations over large arrays

Why Numpy?

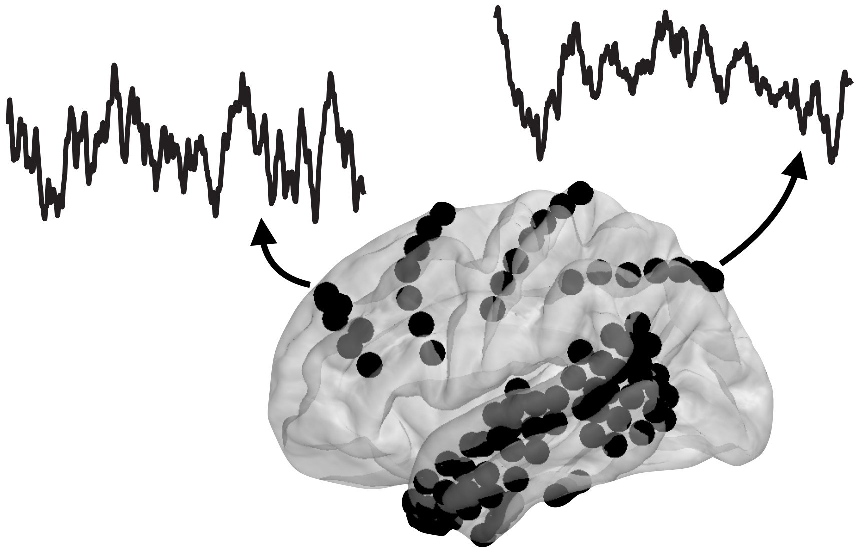
- Provides a large library of functions/methods for performing numerical operations



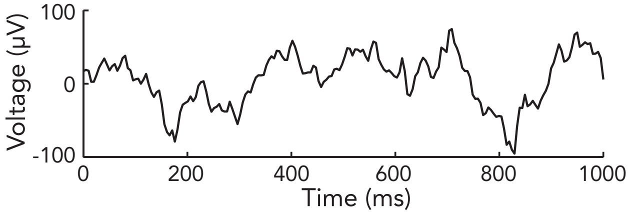
```
In [23]: import numpy as np

In [25]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: np.
```

What kinds of data?

- Multidimensional behavioral and/or neural data

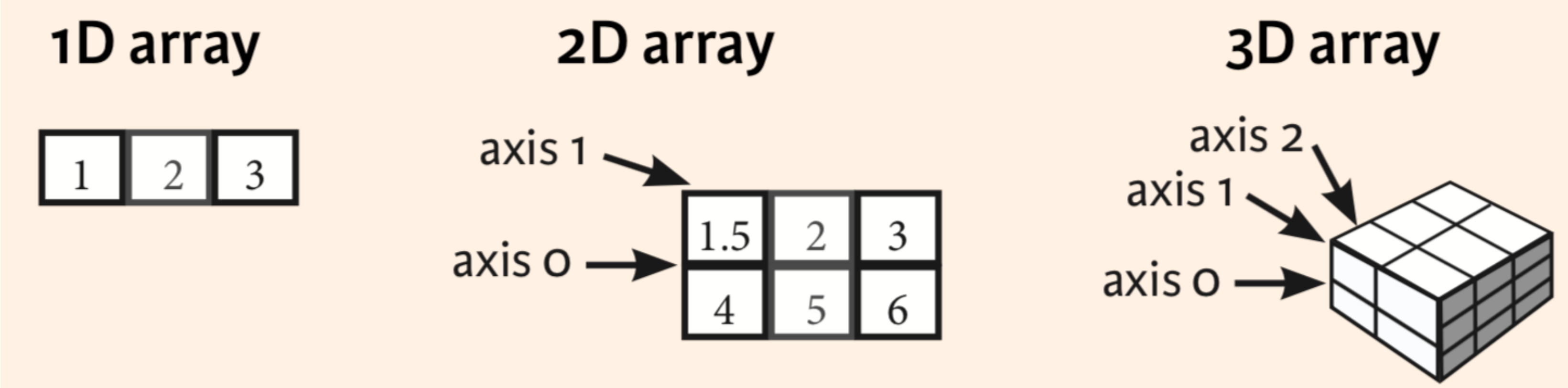


Single-channel recorded signal



Numpy

- Primary numpy object is a multidimensional array containing objects all of the same type

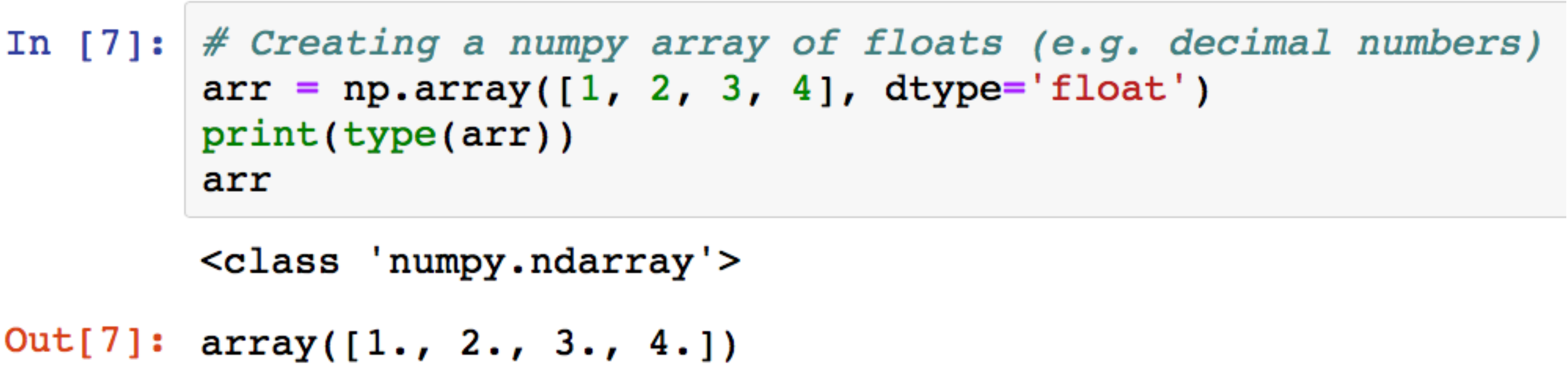


N-D arrays

...

Numpy

- Creating numpy arrays

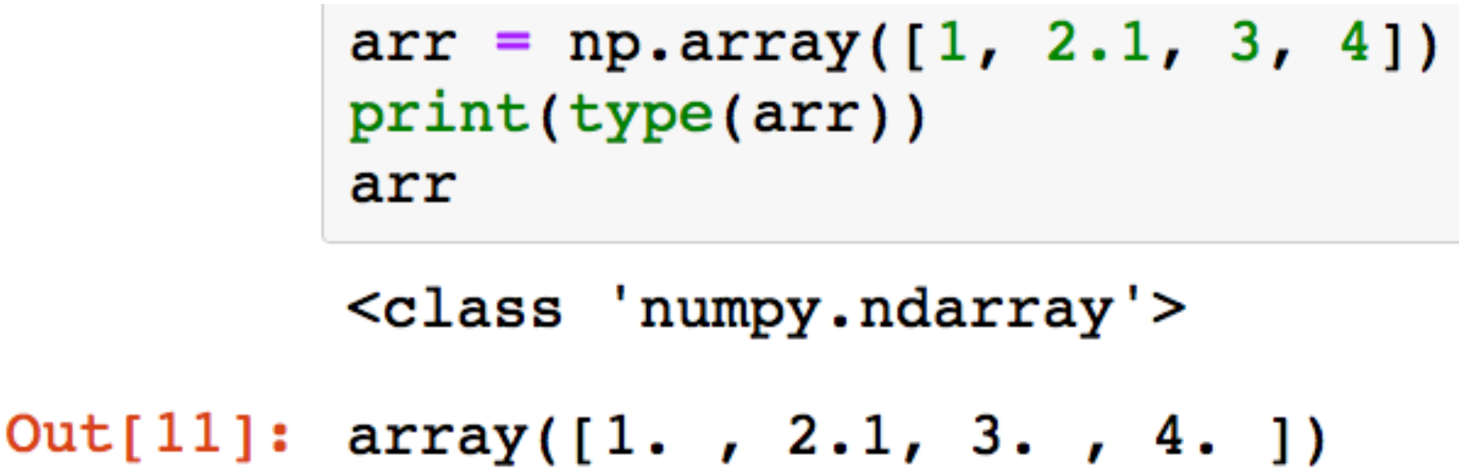


```
In [7]: # Creating a numpy array of floats (e.g. decimal numbers)
        arr = np.array([1, 2, 3, 4], dtype='float')
        print(type(arr))
        arr

        <class 'numpy.ndarray'>

Out[7]: array([1., 2., 3., 4.])
```

Numpy

- Unlike Python lists, the values within numpy arrays must be of the same type

- If you mix types, numpy will try to ‘upcast’ values

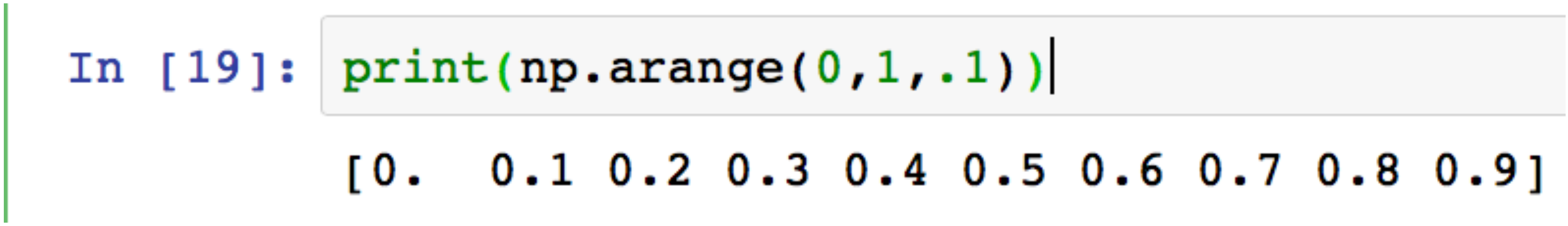


```
arr = np.array([1, 2.1, 3, 4])
print(type(arr))
arr
```

```
<class 'numpy.ndarray'>

Out[11]: array([1. , 2.1, 3. , 4. ])
```

Numpy

- Creating ranges is straightforward with numpy

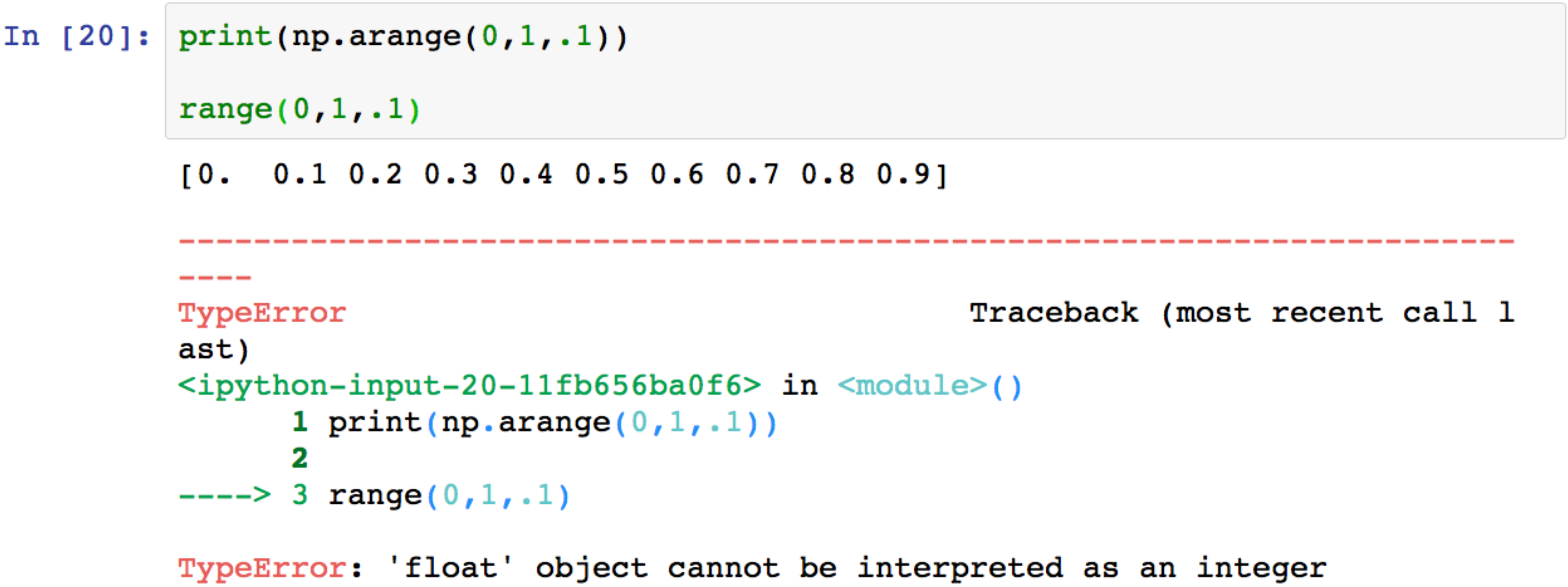


```
In [19]: print(np.arange(0,1,.1))
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

Numpy

- Numpy’s `arange` can be used to create decimal ranges, which are not supported by the standard python `range` function



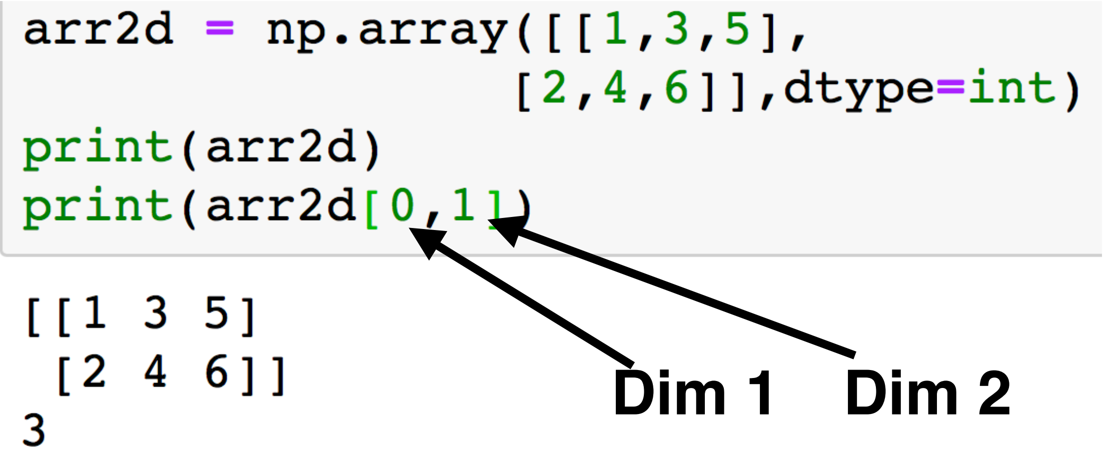
```
In [20]: print(np.arange(0,1,.1))

         range(0,1,.1)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]

---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-20-11fb656ba0f6> in <module>()
      1 print(np.arange(0,1,.1))
      2 
----> 3 range(0,1,.1)

TypeError: 'float' object cannot be interpreted as an integer
```

Numpy

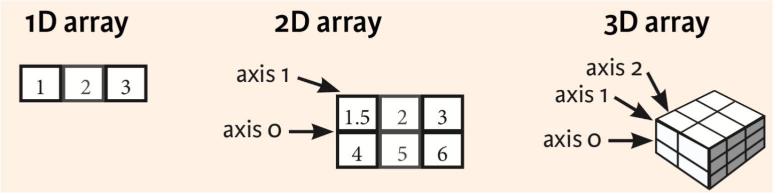
- Indexing elements of a numpy array is similar to indexing elements of python lists

- For **multidimensional** arrays, you index using a set of numbers, where each position in the set corresponds to a dimension of the array



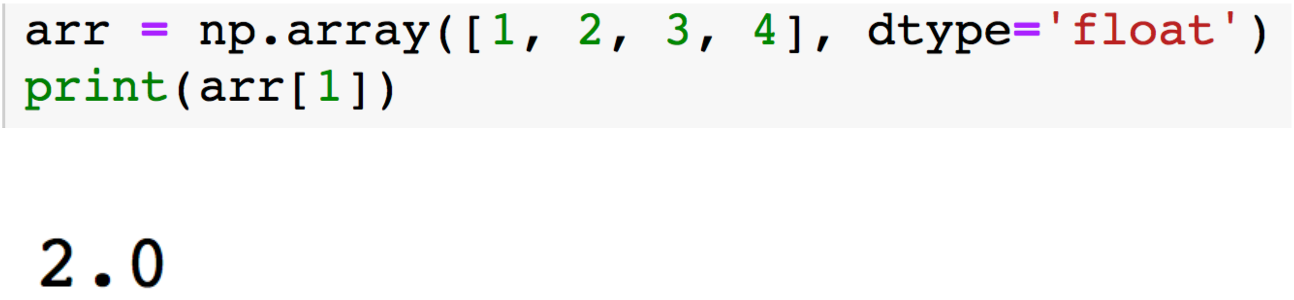
```
arr2d = np.array([[1,3,5],
                  [2,4,6]],dtype=int)
print(arr2d)
print(arr2d[0,1])
```

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



Numpy

- Indexing elements of a numpy array is similar to indexing elements of python lists

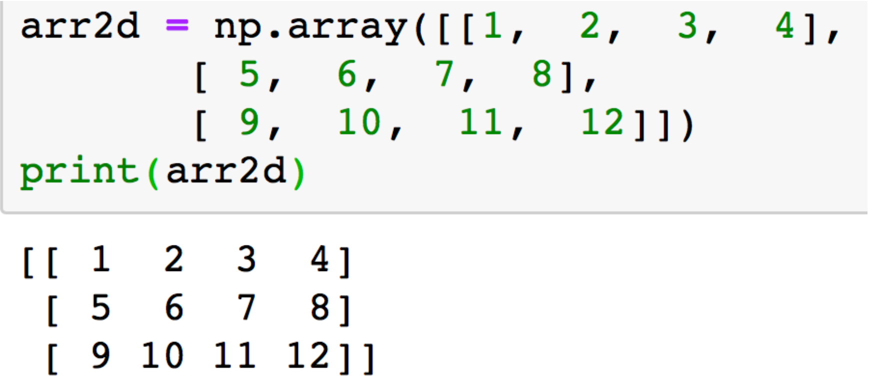


```
arr = np.array([1, 2, 3, 4], dtype='float')
print(arr[1])
```

```
2.0
```

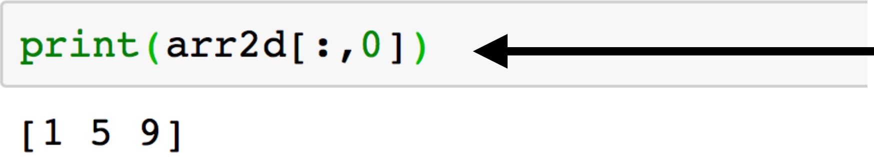
Numpy

- You can also slice arrays to pull out an entire row or column (or other dimension…)



```
arr2d = np.array([[1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9,  10,  11,  12]])
print(arr2d)
```

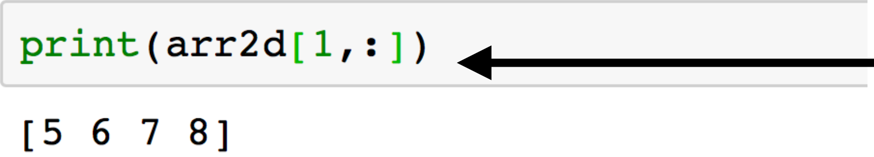
```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```



```
print(arr2d[:,0])
```

```
[1 5 9]
```

Grab all rows, first column



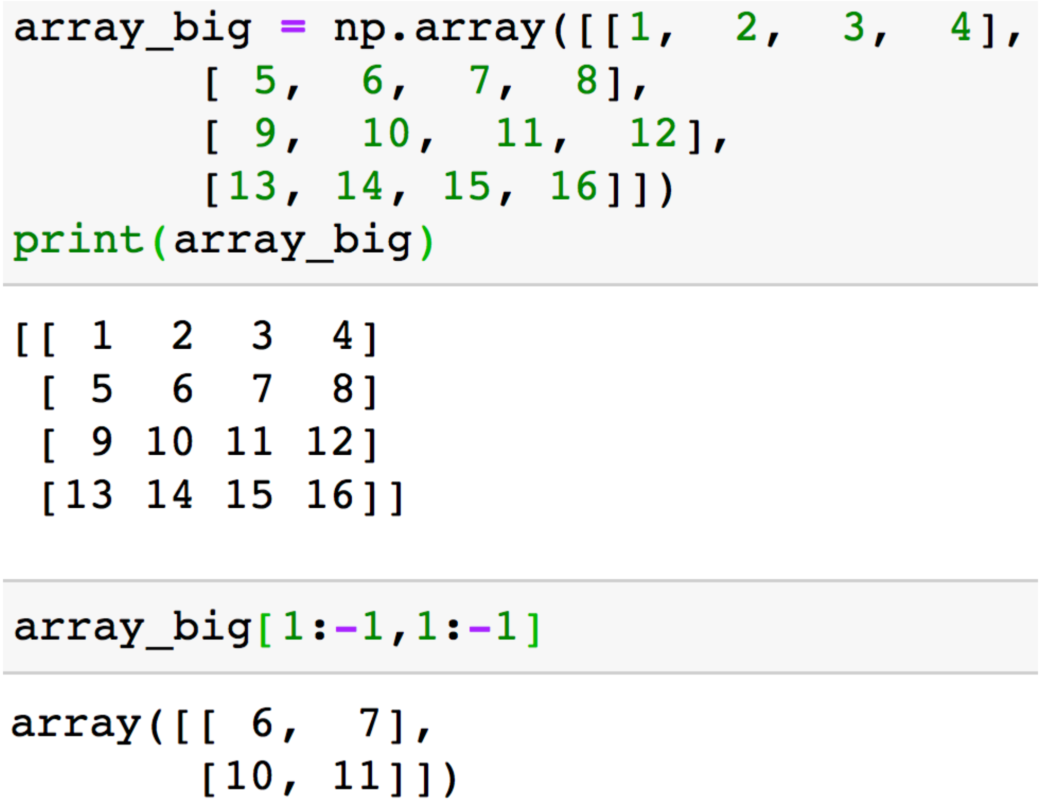
```
print(arr2d[1,:])
```

```
[5 6 7 8]
```

Grab second row, all columns

Numpy

- Use slice notation to access sub-arrays of an array

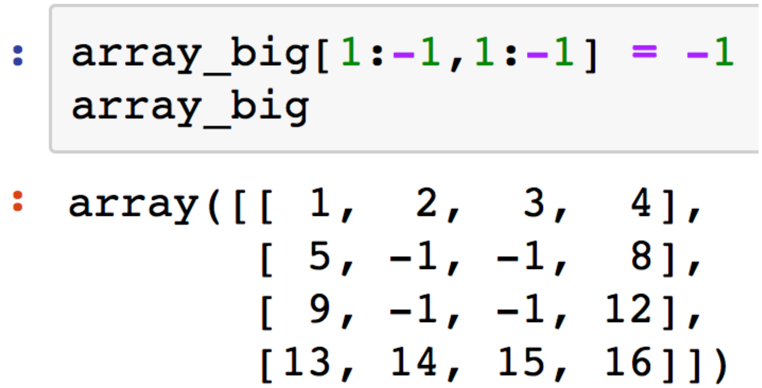


```
array_big = np.array([[1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9,  10,  11,  12],
        [13, 14, 15, 16]])
print(array_big)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

```
array_big[1:-1,1:-1]
```

```
array([[ 6,  7],
       [10, 11]])
```

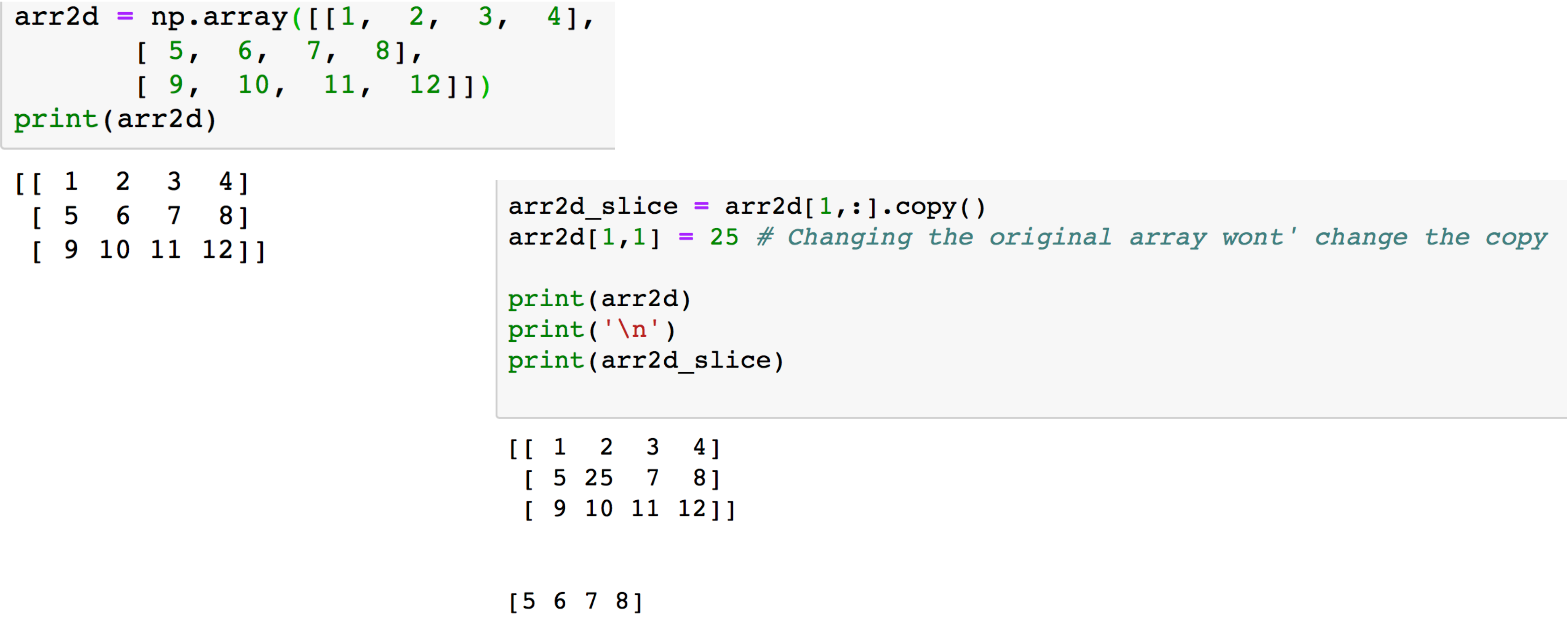


```
: array_big[1:-1,1:-1] = -1
  array_big
```

```
: array([[ 1,  2,  3,  4],
         [ 5, -1, -1,  8],
         [ 9, -1, -1, 12],
         [13, 14, 15, 16]])
```

Numpy

- All of the indexing on the previous slides are 'views'
- If you want to **copy** data into a new object, use `.copy()`



```
arr2d = np.array([[1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9,  10,  11,  12]])
print(arr2d)
```

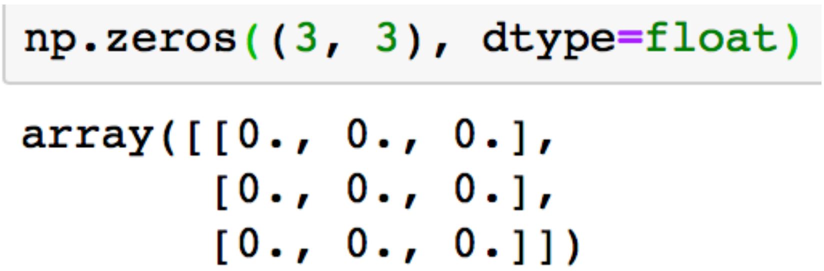
```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
arr2d_slice = arr2d[1,:].copy()
arr2d[1,1] = 25 # Changing the original array wont' change the copy

print(arr2d)
print('\n')
print(arr2d_slice)
```

```
[[ 1  2  3  4]
 [ 5 25  7  8]
 [ 9 10 11 12]]


[5 6 7 8]
```

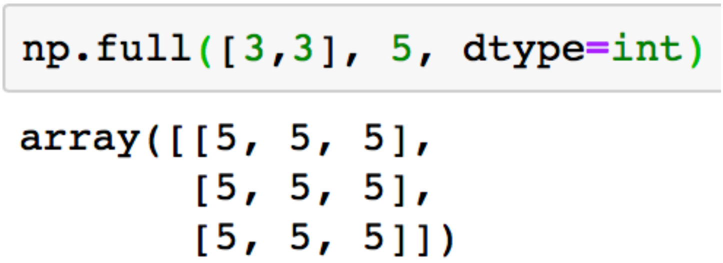
Numpy

- Numpy also has specialized methods for creating arrays filled with specific values



```
np.zeros((3, 3), dtype=float)
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```



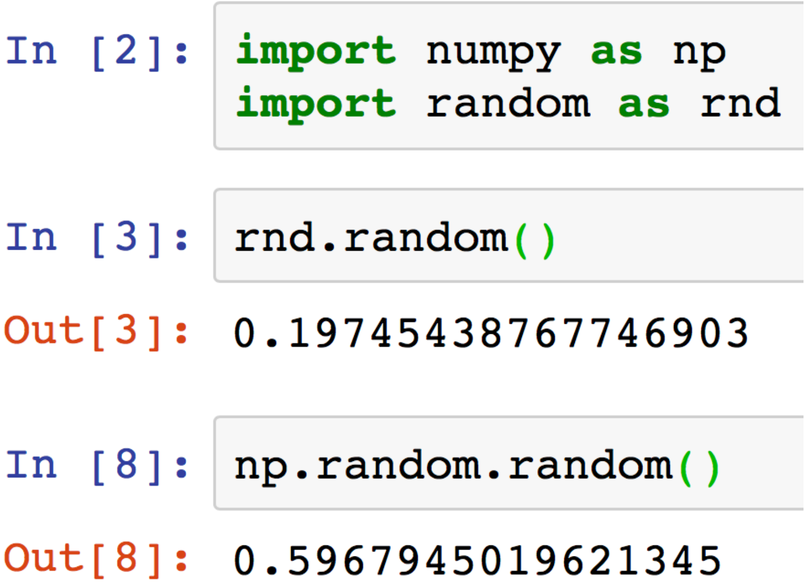
```
np.full([3,3], 5, dtype=int)
```

```
array([[5, 5, 5],
       [5, 5, 5],
       [5, 5, 5]])
```

Random Numbers

- Two libraries for generating random numbers:
  - Numpy
  - Random

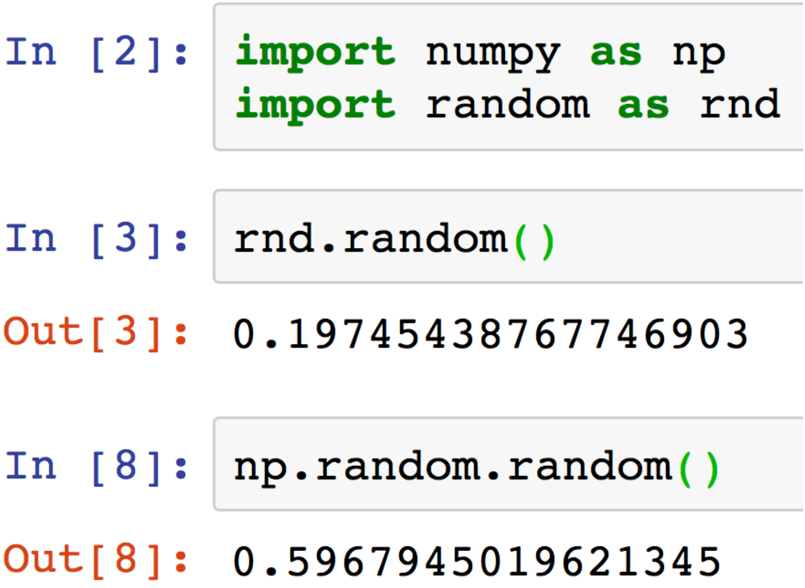
- Both do basically the same thing, Numpy's is a little better for generating arrays of multiple random numbers



```
In [2]: import numpy as np
        import random as rnd

In [3]: rnd.random()

Out[3]: 0.19745438767746903

In [8]: np.random.random()

Out[8]: 0.5967945019621345
```

Random Numbers

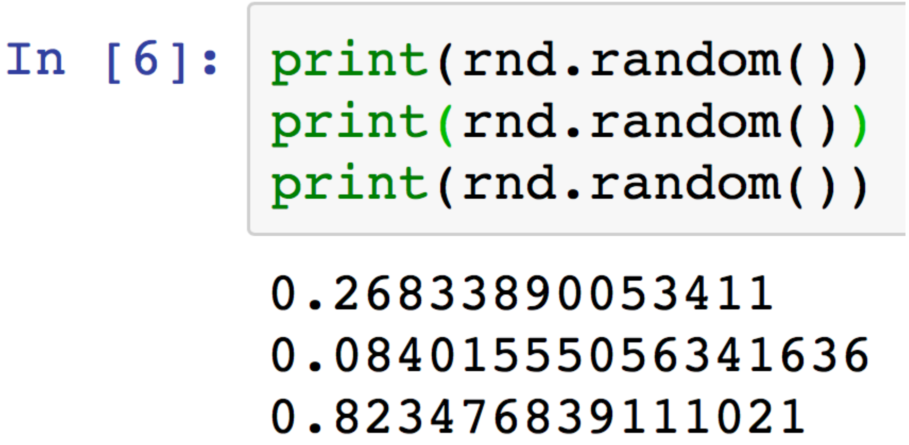
- `rnd.random()` and `np.random.random()` each return a random number between 0 and 1 drawn from a uniform distribution



```
In [2]:  import numpy as np
         import random as rnd

In [3]:  rnd.random()

Out[3]:  0.19745438767746903

In [8]:  np.random.random()

Out[8]:  0.5967945019621345
```

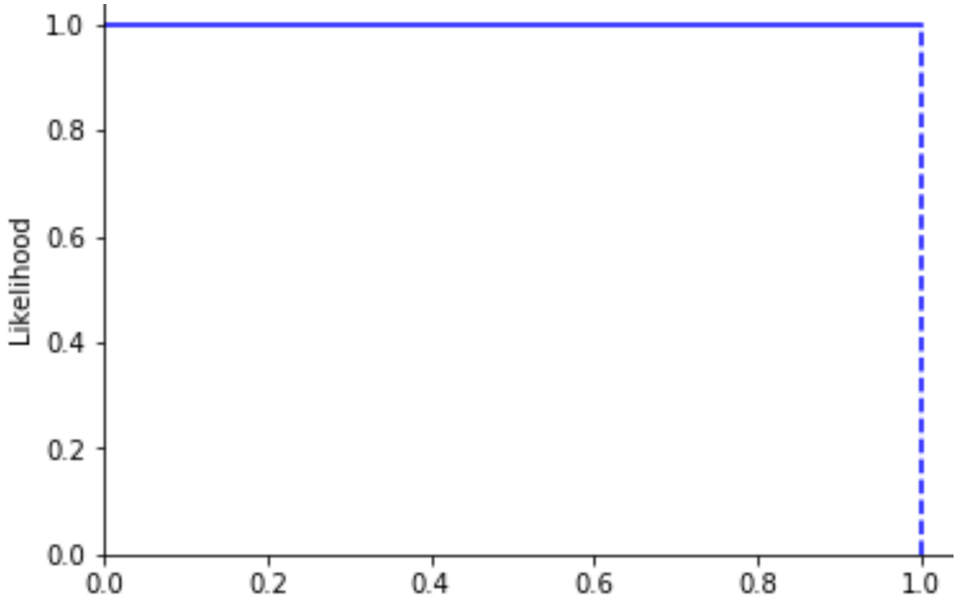
Random Numbers

- Repeated calls to these functions produce more of these random numbers



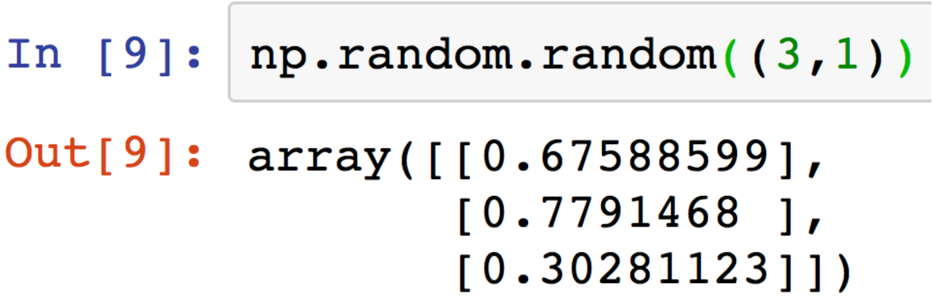
```
In [6]: print(rnd.random())
        print(rnd.random())
        print(rnd.random())

        0.26833890053411
        0.08401555056341636
        0.82347683911021
```

Uniform Distribution

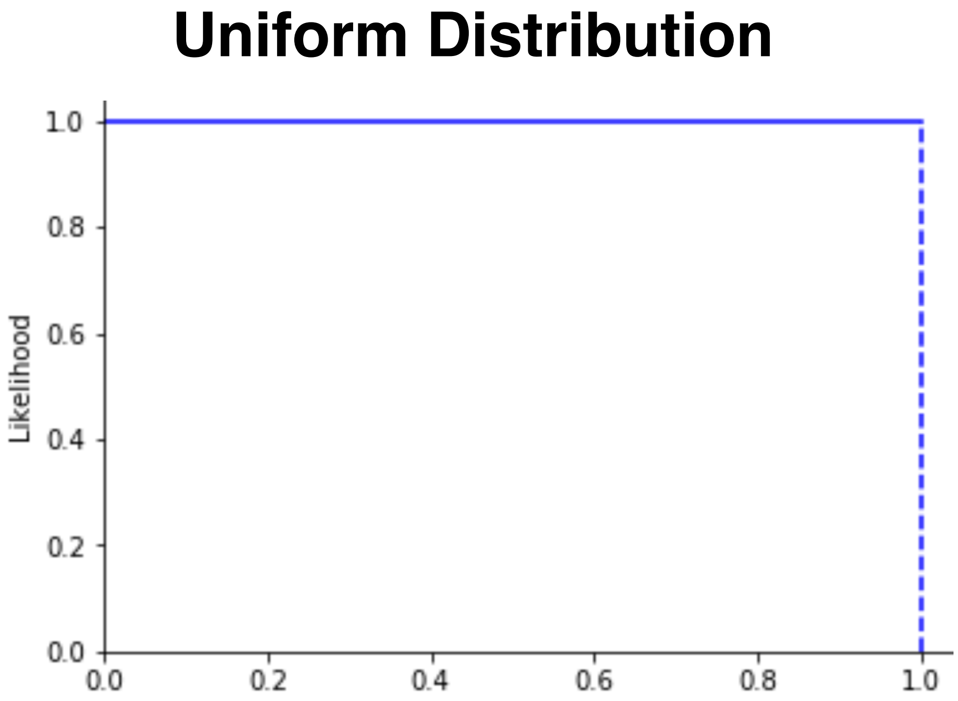


Random Numbers

- You can get an array of random numbers within a single call to the numpy version



```
In [9]:  np.random.random((3,1))

Out[9]:  array([[0.67588599],
                [0.7791468 ],
                [0.30281123]])
```

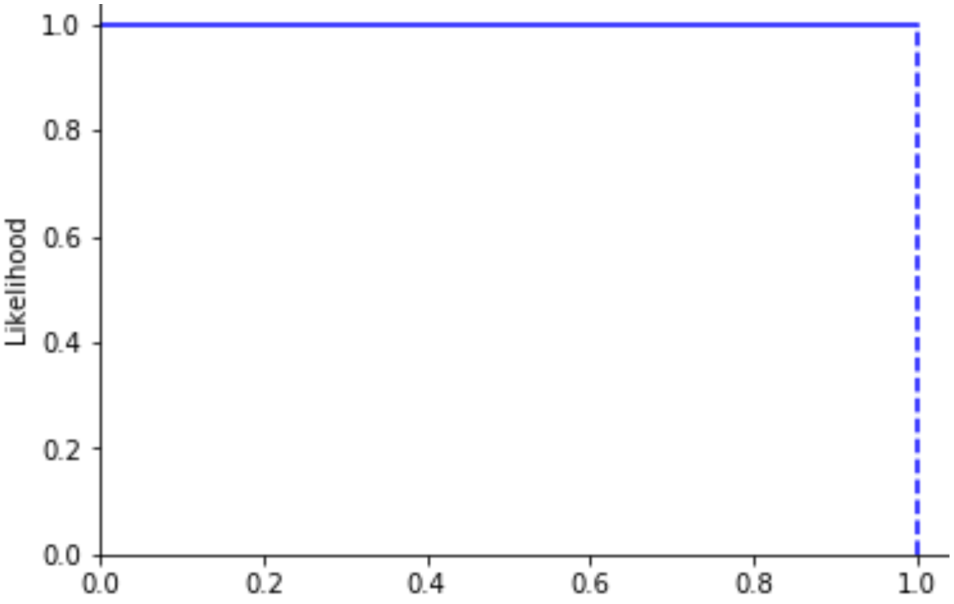


Uniform Distribution

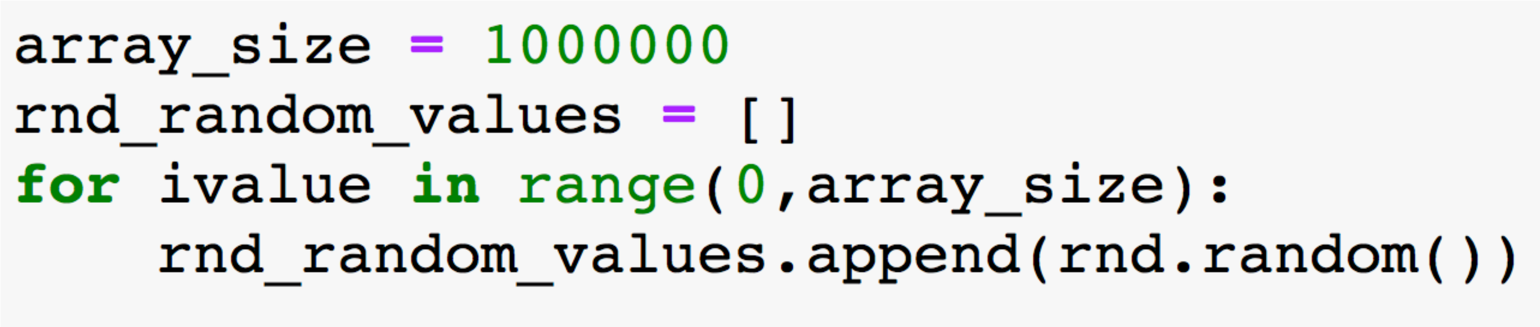
Random Numbers

- How could we confirm for ourselves (empirically) that these random number generators are drawing values from a uniform distribution?
  - rnd.random()?
  - np.random.random?

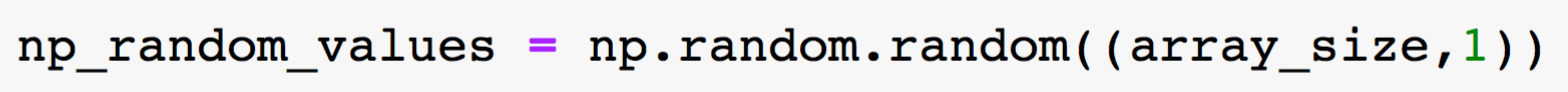
Uniform Distribution



Random Numbers

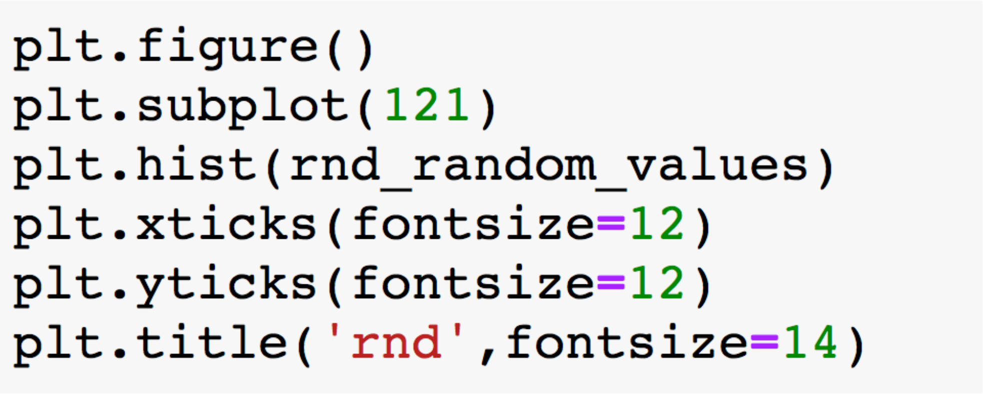


```
array_size = 1000000
rnd_random_values = []
for ivalue in range(0,array_size):
    rnd_random_values.append(rnd.random())
```

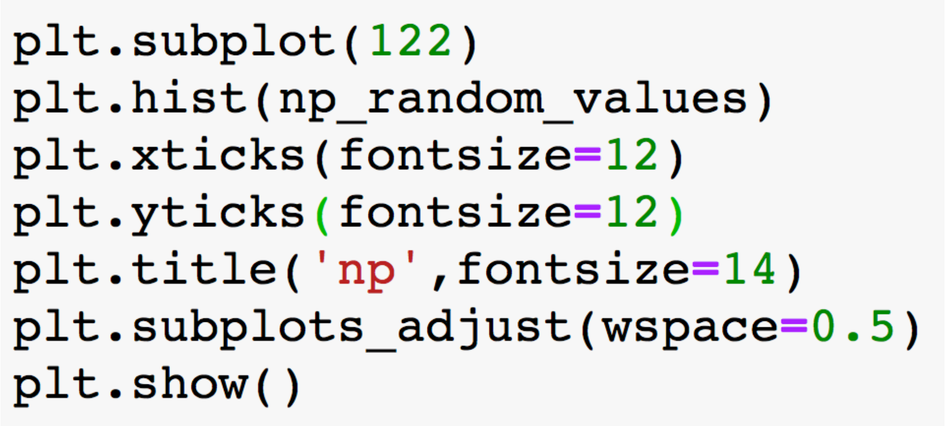


```
np_random_values = np.random.random((array_size,1))
```

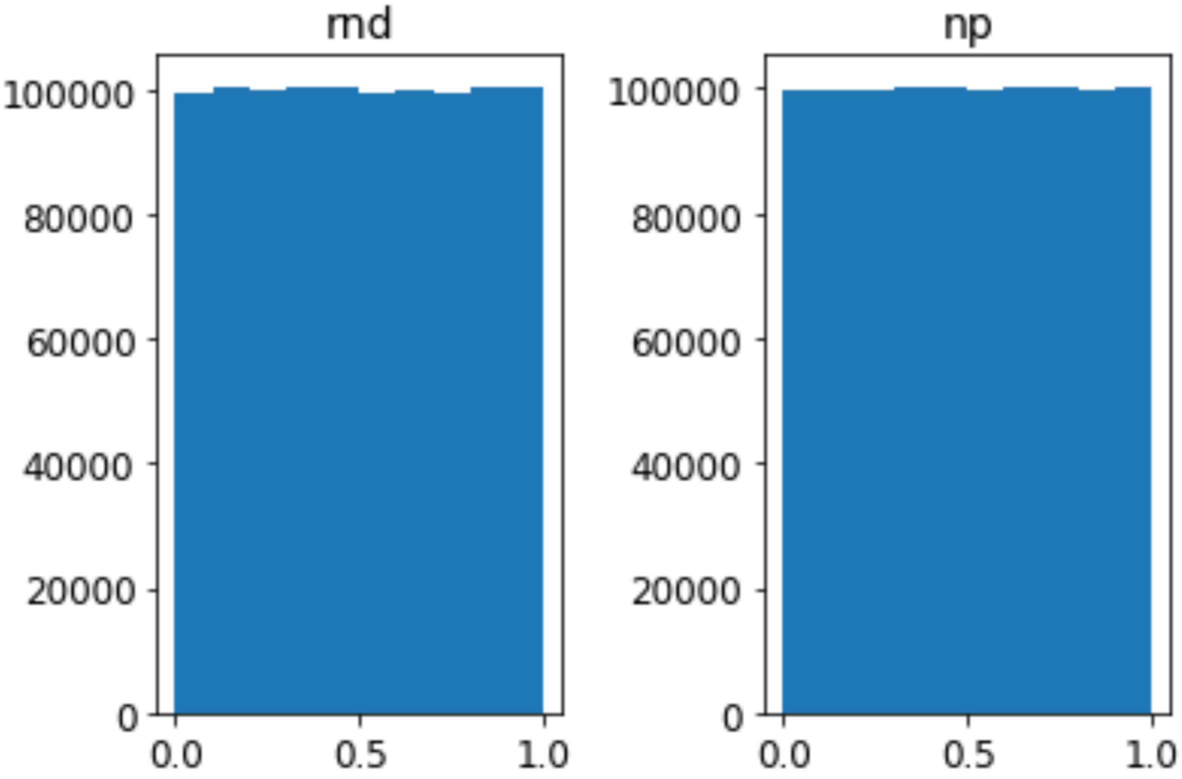
Random Numbers



```
plt.figure()
plt.subplot(121)
plt.hist(rnd_random_values)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.title('rnd',fontsize=14)
```

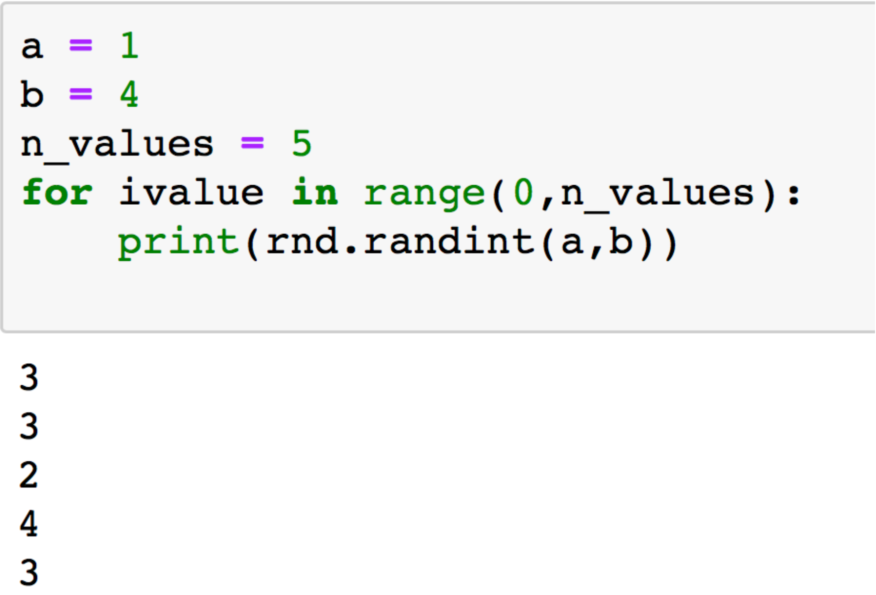


```
plt.subplot(122)
plt.hist(np_random_values)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.title('np',fontsize=14)
plt.subplots_adjust(wspace=0.5)
plt.show()
```



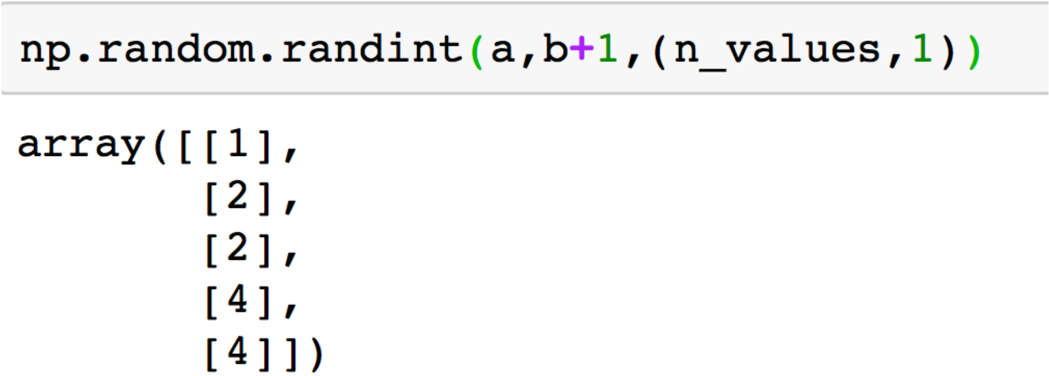
Random Numbers

- There are also functions to generate random integers



```
a = 1
b = 4
n_values = 5
for ivalue in range(0,n_values):
    print(rnd.randint(a,b))
```

```
3
3
2
4
3
```

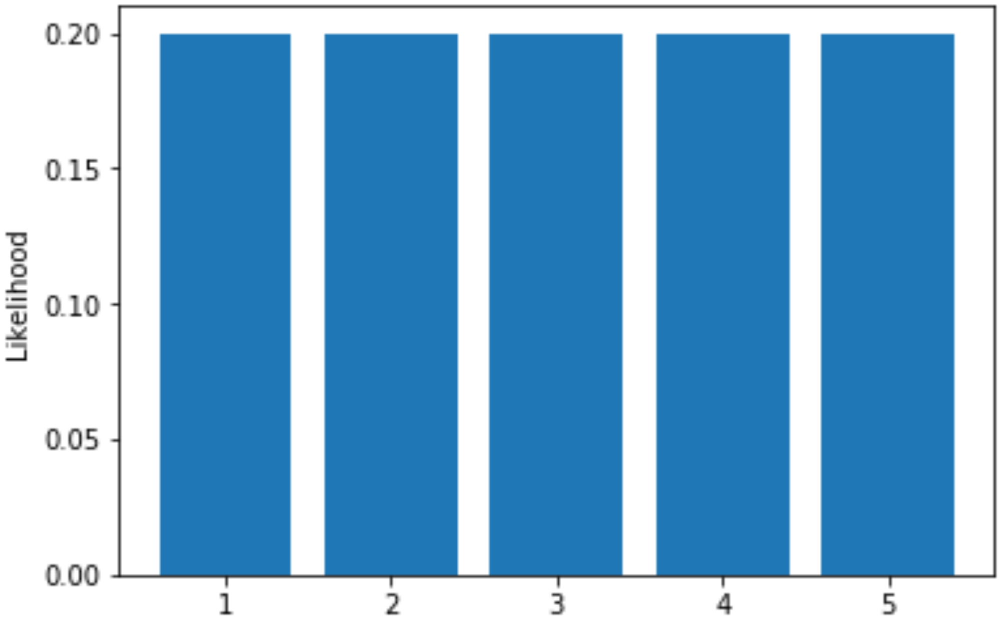


```
np.random.randint(a,b+1,(n_values,1))
```

```
array([[1],
       [2],
       [2],
       [4],
       [4]])
```

Random Numbers

- How could we confirm for ourselves (empirically) that these random number generators are drawing values from a uniform distribution over **discrete integers**?
  - rnd.randint()?
  - np.random.randint?


Likelihood