PS1_solution

April 5, 2021

1 Problem Set 1

This notebook is adapted from the Jupyter-Python-Workshop-STROBE-2018-02-26 repository by Dr. Antonino Ingargiola.

1.1 A Numpy Primer

> *[NumPy] is everywhere. It is all around us. Even now, in this very room. You can see it when you look out your window or when you turn on your television. You can feel it when you go to work … when you go to church … when you pay your taxes.*

```
- Morpheus, The Matrix
```

Source

1.1.1 What is Numpy

Numpy is the core package for arrays computation in Python. In this notebook we will review a few basic concepts on how to use Numpy arrays.

1.1.2 Importing Numpy

Python has only a small number of builtins. All the other functions are organized in packages that need to be imported. Here we import numpy:

[1]:
```
import numpy as np
```

[2]:
```
np.__version__  # most packages have a "version string"
```

[2]:
```
'1.19.2'
```

All the functions provided by numpy are now accessible with the prefix `np.`.

Running a cell: You can run a cell with SHIFT+ENTER. See menu Help -> User Interface Tour for more info.

Autocompletion: Use TAB key to auto-complete commands. Two TAB show the list of alternatives. Autocompletion is a great help in avoiding spelling errors!

1.1.3 About namespaces

The `np` prefix is called a *namespace* and helps avoiding confusion when different packages have a function with the same name. A classical example is the python builtin `max()` and numpy's `max()`. We call the latter typing `np.max()`, so the "namespace" resolves the ambiguity.

Trivia: Can you find out the difference between the builtin max() and np.max()?

1.1.4 Numpy array creation

Manually entering an array:

[3]:
```
np.array([[5, 2, 3],
          [7, 8, 1]])  # NOTE: line splitting here is only for aesthestics
```

[3]:
```
array([[5, 2, 3],
       [7, 8, 1]])
```

Array of zeros:

[4]:
```
np.zeros((3,2))  # NOTE the second set of ()
```

[4]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

Array of random values:

[5]:
```
np.random.random((3,2))  # uniform distribution
```

[5]:
```
array([[0.19366878, 0.098288  ],
       [0.04517833, 0.08385574],
       [0.81875116, 0.88610615]])
```

1.2 Problem 1

- create a 3x4 array ones

1.3 Solution 1

Similar to `np.zeros((n, m))`, we use:

[6]:
```
np.ones((3,4))
```

[6]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

1.4 Problem 2

- Create an array of 10 numbers starting from 0 to 9 using the function `np.arange`
- Create an array of 10 numbers starting from 1 - 10

See also: Array creation cheatsheet

1.5 Solution 2

Using `np.arange` we can make an array from 0 - 9 by realizing that this is a simple array with 10 elements:

[7]:
```
np.arange(10)
```

[7]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

More complex arrays can be also be created using the `np.arange(start, stop, step)` syntax, where `stop` is not inclusive

[8]:
```
np.arange(1, 11, 1)
```

[8]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

1.5.1 Numpy Indexing

Indexing is a way to get one or more elements of the array.

- Index starts at 0, i.e. 0 is the first element
- Index can be negative: -1 is the last element, -2 is the second last, etc…

Scalars We can get one element at time with a scalar index:

[9]:
```
x = np.arange(10)
x
```

[9]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[10]:
```
x[0], x[2], x[-1], x[-2]
```

[10]: (0, 2, 9, 8)

NOTE for MATLAB users: Python uses [ ] when indexing and ( ) when calling a function. MATLAB uses ( ) for both.

Slicing We can also get a "slice" of the array with the following syntax.

- Slice one dimension: [**start : stop : step**]
- You can omit *start*, *stop* or *step* and this will happen:
  1. omitting **start**: slices from the beginning
  2. omitting **stop**: slice till the end
  3. omitting **step**: use step=1

Before running the next cells try to "predict" the output of the following commands - recall that `stop` is not inclusive and that indexing begins at 0

[11]:
```
x[2:10]
```

[11]: array([2, 3, 4, 5, 6, 7, 8, 9])

[12]:
```
x[2:10:3]
```

[12]: array([2, 5, 8])

[13]:
```
x[::]
```

[13]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[14]:
```
x[:]
```

[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[15]:
```
x[::2]
```

[15]: array([0, 2, 4, 6, 8])

[16]:
```
x[2::2]
```

[16]: array([2, 4, 6, 8])

[17]:
```
x[::-1]
```

[17]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

1.6 Problem 3

- Discard the first and last elements in `x`, then take one every 2 of the remaining elements

---

1.7 Solution 3

[18]:
```
x[1:9:2]
```

[18]:
```
array([1, 3, 5, 7])
```

NOTE Unlike in MATLAB, in Python indexing can be chained. For example x[3:-1][::2] is equivalent to x[3:-1:2].

1.8 Problem 4

- Discard the first two elements and the last elements in x, then select every second element
  - get the result with two slices (`x[ ][ ]`)
  - get the result with one slice (`x[ ]`)
- Discard the first two elements and the last elements in x, then select every second element, finally reverse the order

---

1.9 Solution 4

[19]:
```
# both of the following work
x[2:9][::2]
x[2:-1][::2]
```

[19]:
```
array([2, 4, 6, 8])
```

[20]:
```
x[2:9:2]
```

[20]:
```
array([2, 4, 6, 8])
```

[21]:
```
x[2:9:2][::-1]
```

[21]:
```
array([8, 6, 4, 2])
```

Boolean mask `numpy` also uses comparison operators such as `<` and `>` as element-wise `ufuncs` (vectorized operations in `numpy`).

Other operators are: - `==` equal (defined as, exactly equal to) - `!=` not equal to - `<=` less than or equal to - `>=` greater than or equal to

The result of these comparison operators is always an array with a Boolean data type, i.e. `True` or `False`.

For example, get all elements in `x` larger than 5:

[22]:
```
x > 5
```

[22]:
```
array([False, False, False, False, False, False,  True,  True,  True,
        True])
```

[23]:
```
x[x > 5]
```

[23]:
```
array([6, 7, 8, 9])
```

Boolean masks can be negated with `~`, combined with `*` (**AND**) or `+` (**OR**) or compared with `==`.

For example:

[24]:
```
(~(x > 5))*(~(x < 7)) == ~((x > 5)+(x < 7))
```

[24]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True])
```

The previous expression always returns all True, for any `x`. This is called the De Morgan Law in boolean logic.

[25]:
```
all((~(x > 5))*(~(x < 7)) == ~((x > 5)+(x < 7)))
```

[25]:
```
True
```

1.10 Problem 5

- Create an array `y` of 10 random numbers in [0..1], then select all the elements between 0.2 and 0.7:

1.11 Solution 5

[26]:
```
y = np.random.random(10)
y
```

[26]:
```
array([0.9512791 , 0.689771  , 0.91847576, 0.56465696, 0.42325581,
       0.07755999, 0.65439637, 0.43370594, 0.51442547, 0.94570238])
```

[27]:
```
y[(y > 0.1)*(y < 0.8)]
```

[27]:
```
array([0.689771  , 0.56465696, 0.42325581, 0.65439637, 0.43370594,
       0.51442547])
```

1.11.1 2D Arrays

Numpy array can have multiple dimensions. Here it is a 2D array (it will be indexed by row, column):

[28]:
```
A = np.arange(20)
A
```

[28]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

[29]:
```
A = np.arange(20).reshape(5, 4)
A
```

[29]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

[30]:
```
A.shape
```

[30]:
```
(5, 4)
```

Indexing rules

- Index: [**rows, cols**]
- **row** or **cols** can be scalars, slices or arrays
- Trailing dimension (`cols`) can be omitted

  note that even for row and columns, the index starts at 0!

[31]:
```
A[1,2]
```

[31]:
```
6
```

[32]:
```
A[:2, :3]
```

[32]:
```
array([[0, 1, 2],
       [4, 5, 6]])
```

[33]:
```
A[0, :]
```

[33]:
```
array([0, 1, 2, 3])
```

[34]:
```
A[0]
```

[34]:
```
array([0, 1, 2, 3])
```

[35]:
```
A[0,0]
```

[35]:
```
0
```

[36]:
```
A[A > 5]
```

[36]:
```
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

[37]:
```
A[::-1]
```

[37]:
```
array([[16, 17, 18, 19],
       [12, 13, 14, 15],
       [ 8,  9, 10, 11],
       [ 4,  5,  6,  7],
       [ 0,  1,  2,  3]])
```

1.12 Problem 6

- create a 4x12 array with horizontal layout where the first row goes from 0 to 11 (left to right)
- now convert the array you just created to verticle Hint: how do you transpose an array?

1.13 Solution 6

[38]:
```
spots = np.arange(48).reshape(4, 12)
spots
```

[38]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
       [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]])
```

[39]:
```
spots.T
```

[39]:
```
array([[ 0, 12, 24, 36],
       [ 1, 13, 25, 37],
       [ 2, 14, 26, 38],
       [ 3, 15, 27, 39],
       [ 4, 16, 28, 40],
       [ 5, 17, 29, 41],
       [ 6, 18, 30, 42],
       [ 7, 19, 31, 43],
       [ 8, 20, 32, 44],
       [ 9, 21, 33, 45],
       [10, 22, 34, 46],
       [11, 23, 35, 47]])
```

1.14 Problem 7

Selecting Random Points

- What is the shape of the following 2D matrix?
- Make a scatter plot of the data
- use fancy indexing to select 20 random points
  - Hint: first choose 20 random indices with no repeats using `np.random.choice()`, then use these indices to select a portion of the original array

[40]:
```
import matplotlib.pyplot as plt
import seaborn
```

[41]:
```
%matplotlib inline
seaborn.set()  # for plot styling
```

[42]:
```
# define a random state instance
rand = np.random.RandomState(42)
```

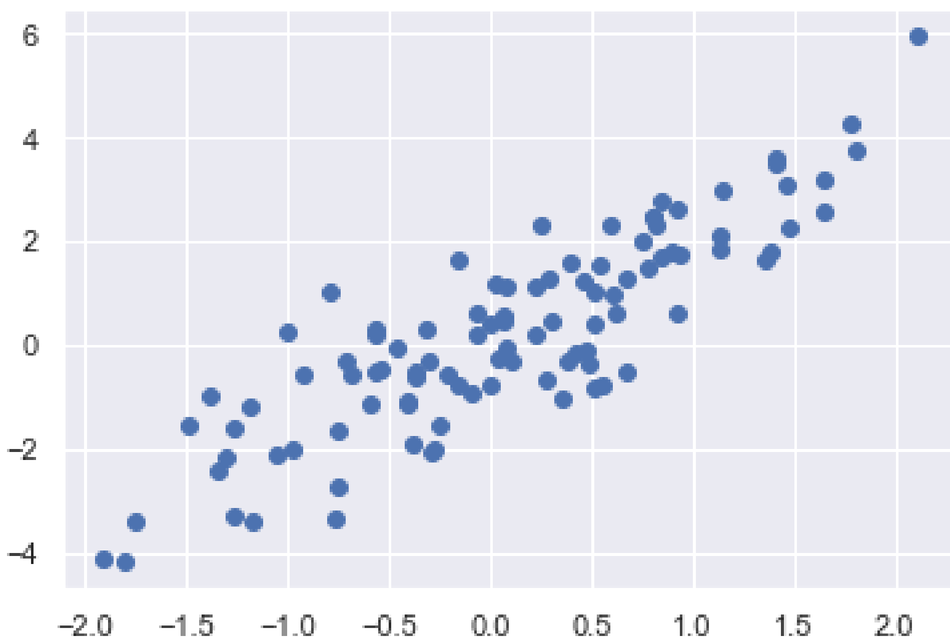
[43]:
```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
```

1.15 Solution 7

[44]:
```
X.shape
```

[44]: (100, 2)

[45]:
```
plt.scatter(X[:, 0], X[:, 1]);
```



[46]:
```
# define indices with no repeats by passing `replace=False`
indices = np.random.choice(X.shape[0], 20, replace=False)
indices
```

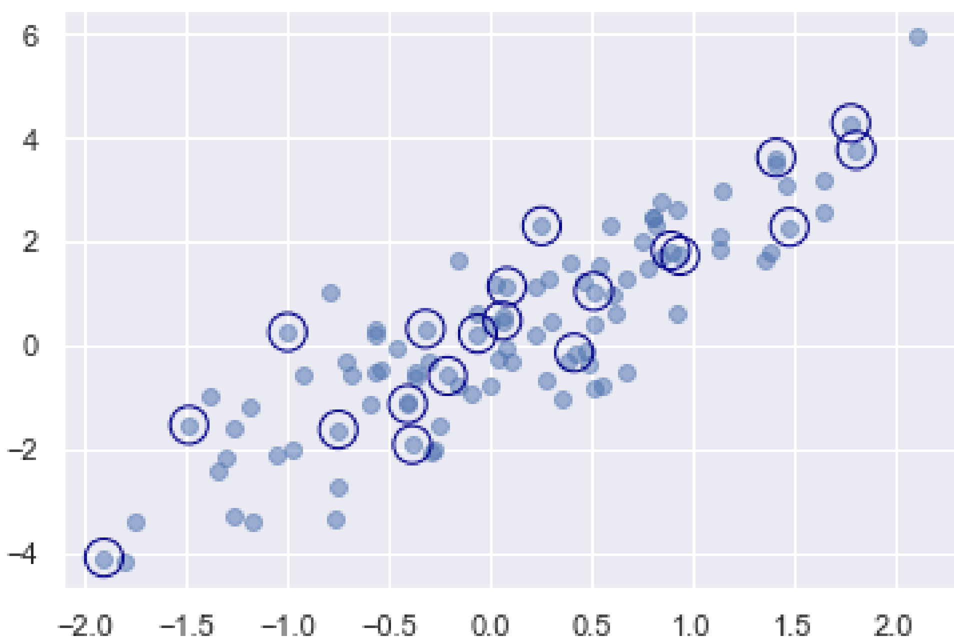
[46]:
```
array([33, 55, 86,  7, 54, 78, 95, 56,  0, 83, 42, 28, 16, 30, 20, 71, 67,
       60, 31, 57])
```

[47]:
```
# index the X array
selection = X[indices]
selection
```

```
[47]: array([[-0.31750674,  0.31968711],
       [ 1.78285987,  4.27550608],
       [-0.05932052,  0.22604784],
       [ 1.8087944 ,  3.75820034],
       [-0.20945632, -0.58625196],
       [-1.90507889, -4.08639424],
       [ 0.0847964 ,  1.13167598],
       [-0.9982874 ,  0.25611506],
       [-0.40599258, -1.129809  ],
       [-1.48535581, -1.5328922 ],
       [ 0.9389648 ,  1.72376364],
       [ 0.89366635,  1.82281235],
       [ 0.41723826, -0.1375558 ],
       [ 0.51374789,  1.03934128],
       [-0.74783396, -1.6199432 ],
       [ 1.41446454,  3.61466573],
       [ 0.25630301,  2.29640712],
       [-0.38281071, -1.9085007 ],
       [ 1.4798887 ,  2.27800264],
       [ 0.06232119,  0.47684914]])
```

Now, let's circle the points we selected on the plot:

```
[48]: plt.scatter(X[:, 0], X[:, 1], alpha=0.5)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', edgecolor='darkblue', s=200);
```



1.15.1 Binning data by indexing

Indexing can be used to bin data manually.

[49]:
```
np.random.seed(42)
x = np.random.randn(100)

# compute a histogram by hand
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)
counts
```

[49]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.])
```

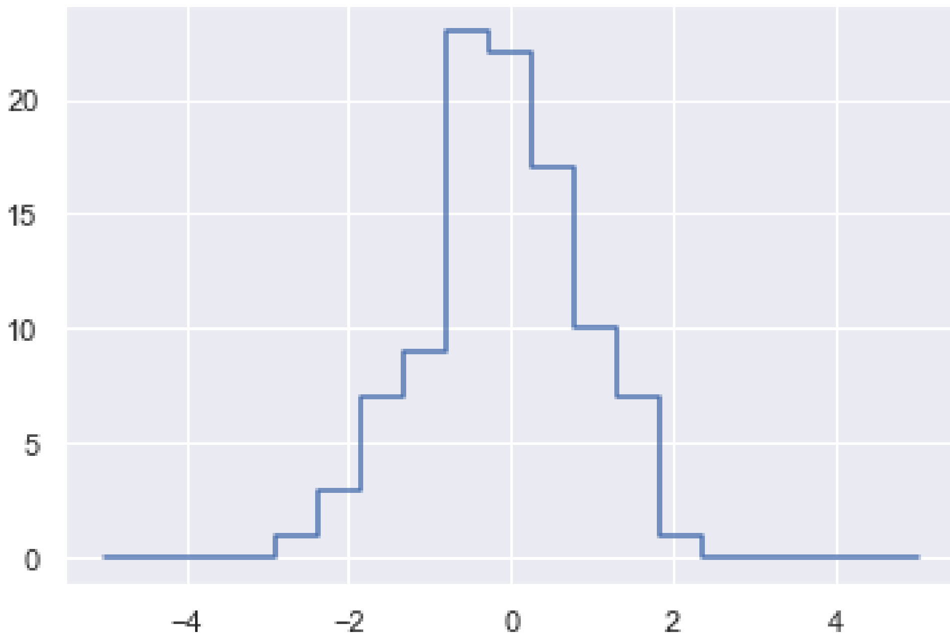
[50]:
```
# find the appropriate bin for each x
i = np.searchsorted(bins, x)
i
```

[50]:
```
array([11, 10, 11, 13, 10, 10, 13, 11,  9, 11,  9,  9, 10,  6,  7,  9,  8,
       11,  8,  7, 13, 10, 10,  7,  9, 10,  8, 11,  9,  9,  9, 14, 10,  8,
       12,  8, 10,  6,  7, 10, 11, 10, 10,  9,  7,  9,  9, 12, 11,  7, 11,
        9,  9, 11, 12, 12,  8,  9, 11, 12,  9, 10,  8,  8, 12, 13, 10, 12,
       11,  9, 11, 13, 10, 13,  5, 12, 10,  9, 10,  6, 10, 11, 13,  9,  8,
        9, 12, 11,  9, 11, 10, 12,  9,  9,  9,  7, 11, 10, 10, 10])
```

[51]:
```
# add 1 to each of these bins
np.add.at(counts, i, 1)
counts
```

[51]:
```
array([ 0.,  0.,  0.,  0.,  0.,  1.,  3.,  7.,  9., 23., 22., 17., 10.,
        7.,  1.,  0.,  0.,  0.,  0.,  0.])
```

[52]:
```
# plot the results
plt.plot(bins, counts, linestyle='-', drawstyle='steps');
```

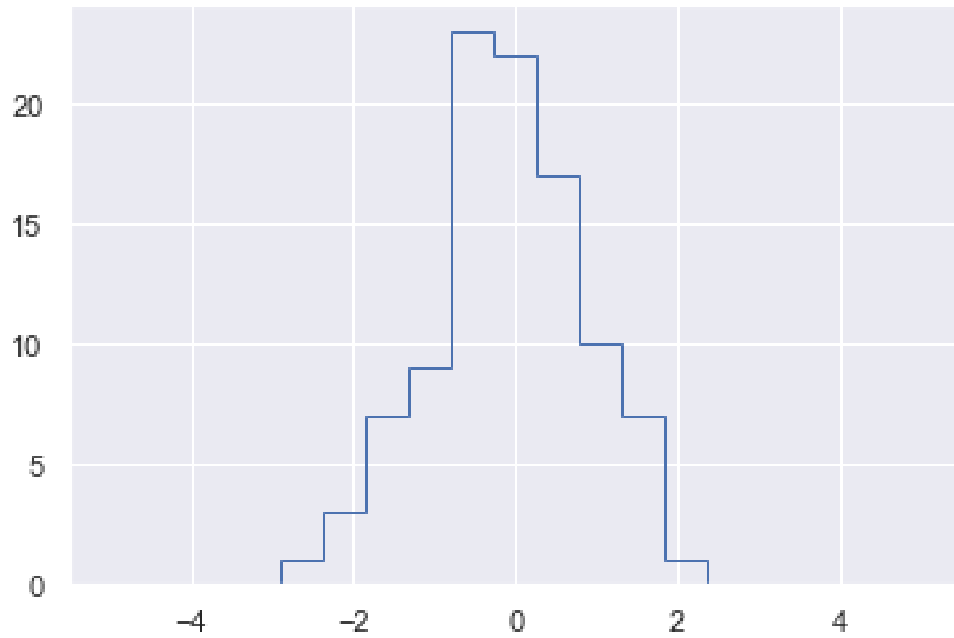


1.16 Problem 8

- create the same plot in one line using `plt.hist()`

1.17 ## Solution 8

[53]:
```
plt.hist(x, bins, histtype='step');
```



1.17.1 Data manipulation with Pandas

[54]:
```
import pandas as pd
pd.__version__
```

[54]: '1.2.3'

[55]:
```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

[55]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

[56]:
```python
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
```

[56]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

[57]:
```python
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

[57]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

[58]:
```python
# list indices of the dataframe
states.index
```

[58]:
```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'],
dtype='object')
```

[59]:
```python
# list columns of the datafram
states.columns
```

[59]:
```
Index(['population', 'area'], dtype='object')
```

[60]:
```python
# show population (for index, in this case states)
# using dataframe attribute feature
states.population
```

[60]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
Name: population, dtype: int64
```

[61]:
```python
# the same thing can be achieved using the dictionary feature
states['population']
```

[61]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
Name: population, dtype: int64
```

1.18 Problem 9

As with `numpy` arrays, DataFrames support array broadcasting. For example, `df['new_colum'] = df['col_1'] * df['col_2']`. Note, that you should try to avoid column assignment via attribute (i.e., use `data['population'] = z` rather than `data.population = z`).

- create a new column for `states` called `pop_density`

[62]:
```
states['pop_density'] = states['population'] / states['area']
states
```

[62]:

	population	area	pop_density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121
Illinois	12882135	149995	85.883763

DataFrames as 2D arrays

[63]:
```
states.values
```

[63]:
```
array([[3.83325210e+07, 4.23967000e+05, 9.04139261e+01],
       [2.64481930e+07, 6.95662000e+05, 3.80187404e+01],
       [1.96511270e+07, 1.41297000e+05, 1.39076746e+02],
       [1.95528600e+07, 1.70312000e+05, 1.14806121e+02],
       [1.28821350e+07, 1.49995000e+05, 8.58837628e+01]])
```

[64]:
```
states.values.shape
```

[64]:
```
(5, 3)
```

Many array-like opercations can be performed. For example we can transpose the full DataFrame to swap rows and columns,

[65]:
```
states.T
```

[65]:

```
             California         Texas      New York       Florida  \
population   3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07
area         4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05
pop_density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02

                 Illinois
population   1.288214e+07
area         1.499950e+05
pop_density  8.588376e+01
```

1.19 Problem 10

- Use `.iloc[]` to select the first three rows and 2 colums by indexing the transposed `states` DataFrame
- Use `.loc[]` to select the population of all the states by indexing with the explicit index and column names

---

1.20 Solution 10

[66]:

```python
states.T.iloc[:3, :2]
```

[66]:

```
             California         Texas
population   3.833252e+07  2.644819e+07
area         4.239670e+05  6.956620e+05
pop_density  9.041393e+01  3.801874e+01
```

[67]:

```python
states.T.loc[:'population', :'Illinois']
```

[67]:

```
             California       Texas    New York     Florida    Illinois
population   38332521.0  26448193.0  19651127.0  19552860.0  12882135.0
```

Boolean masks with DataFrames We can apply Boolean masks to index DataFrames. For example,

[68]:

```python
states[states.pop_density > 100]
```

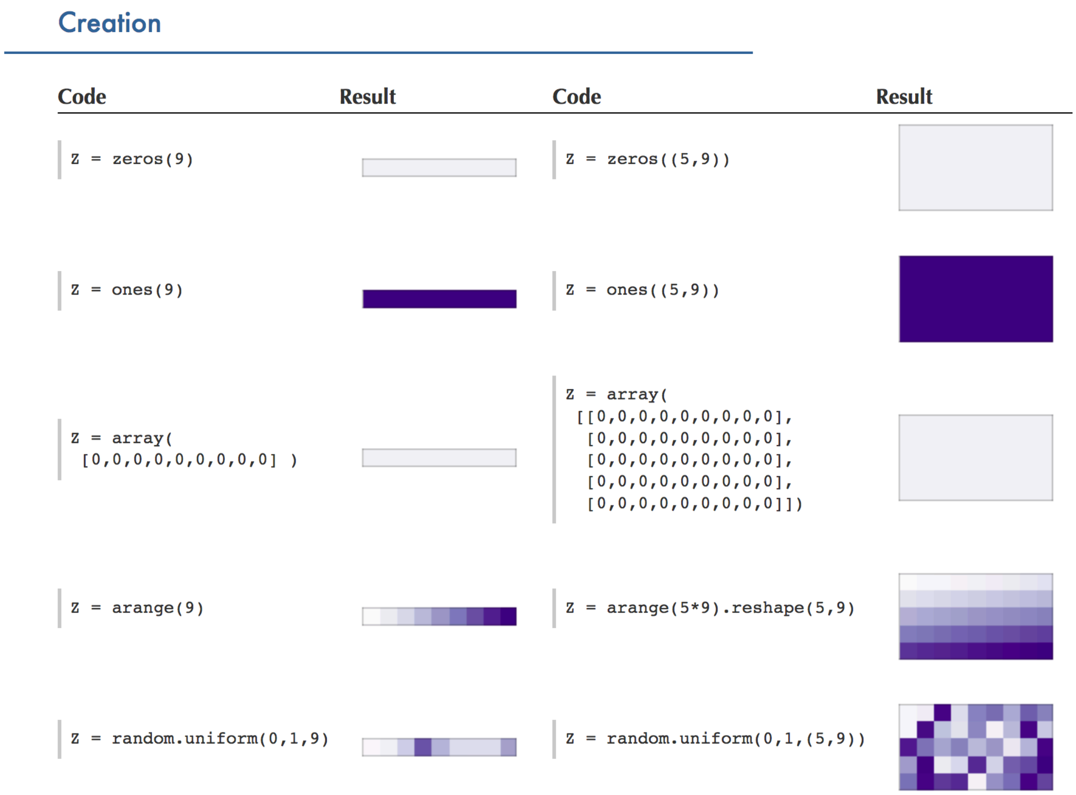
[68]:

```
          population    area  pop_density
New York    19651127  141297   139.076746
Florida     19552860  170312   114.806121
```

COMPLETION If you mastered all the code above you are now a powerful apprentice! You are ready for the workshop. If you want, challenge yourself you'll find one more exercise below!

1.20.1 Numpy Cheatsheets

Array creation In the following cheatsheet the `np.` prefix is omitted. Does the following make sense?



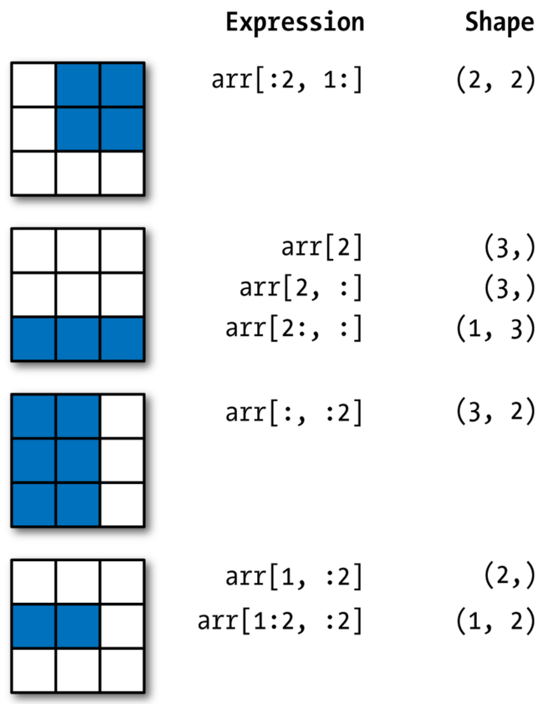
Creation

Code	Result	Code	Result
`Z = zeros(9)`		`Z = zeros((5,9))`	
`Z = ones(9)`		`Z = ones((5,9))`	
`Z = array( [0,0,0,0,0,0,0,0,0] )`		`Z = array( [[0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0]])`	
`Z = arange(9)`		`Z = arange(5*9).reshape(5,9)`	
`Z = random.uniform(0,1,9)`		`Z = random.uniform(0,1,(5,9))`	

>

Source: Numpy Tutorial by *Nicolas P. Rougier*.

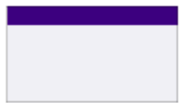
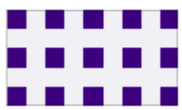
Slicing cheatsheets 1 At this point, you should be able to understand this:



>

Source: **Python for Data Analysis** by *Wes McKinney*, Ch4. NumPy Basics: Arrays and Vectorized Computation.

Slicing cheatsheets 2 A little bit more of "slicing" fun:

Code	Result	Code	Result
`Z`		`Z[...] = 1`	
`Z[1,1] = 1`		`Z[:,0] = 1`	
`Z[0,:] = 1`		`Z[2:,2:] = 1`	
`Z[:,::2] = 1`		`Z[::2,:] = 1`	
`Z[:-2,:-2] = 1`		`Z[2:4,2:4] = 1`	
`Z[::2,::2] = 1`		`Z[3::2,3::2] = 1`	

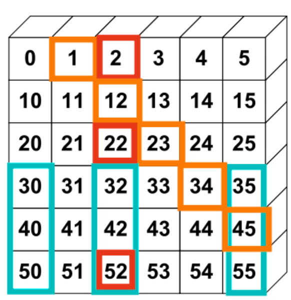
>

Source: Numpy Tutorial by *Nicolas P. Rougier*.

1.21 Challenge question

Advanced indexing

- Given the 2D array `a` in the figure, can you index `a` to obtain the 3 selections highlighted by different colors?



>

Source: **Scipy Lecture Notes** by *Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux, and Pauli Virtanen*. Chapter: The Numpy Arrays Object

[69]:
```python
# Here I create `a` for you
a = np.arange(0, 51, 10)[:,np.newaxis] + np.arange(6)  # broadcasting trick
a
```

[69]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

[ ]:
```python

```

[ ]:
```python

```

2 References

2.1 Basic and Intermediate

- Numpy Tutorial by *Nicolas P. Rougier*

  Get hooked with Numpy by simulating the game of life,

  solve some one-line numpy trivia,

  or, skip to the Quick Reference section for great graphical examples of Numpy's indexing.

- NumPy: creating and manipulating numerical data by *Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux, and Pauli Virtanen*

  Chapter about Numpy from the famous **Scipy Lecture Notes** book.

2.2 Advanced

This is more advanced material not covered in the workshop:

- Elegant Scipy - Ch 1 by *Harriet Dashnow, Stéfan van der Walt, Juan Nunez-Iglesias*

  This free chapter of the *Elegant Scipy* book shows the power and elegance of Numpy by analyzing gene-expression data.

- https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing

  The official reference of advanced indexing

- https://jakevdp.github.io/PythonDataScienceHandbook/02.07-fancy-indexing.html

  Good explanation of fancy indexing using 2D arrays as examples.

- https://github.com/stefanv/teaching/blob/master/2010_scipy_numpy_kittens_dragons/kittens_dragons_

Array broadcasting explained with figures, plus the classical "Jack's problem" from the mailing list

- https://stackoverflow.com/questions/11942747/numpy-multi-dimensional-array-indexing-swaps-axis-order

  Why axis are reordered when fancy indexing is mixed with basic indexing?

2.3 Hinc Sunt Leones

- Einstein Summation in Numpy