EECS 445: Python Tutorial

Presented by: Zhao Fu

September 12, 2016

References:

1. https://docs.python.org/3/tutorial/ (https://docs.python.org/3/tutorial/)
2. https://docs.python.org/3/library/ (https://docs.python.org/3/library/)
3. http://cs231n.github.io/python-numpy-tutorial/ (http://cs231n.github.io/python-numpy-tutorial/)
4. https://github.com/donnemartin/data-science-ipython-notebooks (https://github.com/donnemartin/data-science-ipython-notebooks)

Why Python?

- Easy to learn
- High-level data structures
- Elegant syntax
- Lots of useful packages for machine learning and data science

Install

- https://www.continuum.io/downloads (https://www.continuum.io/downloads)

  conda

Now we have python3 installed

- numpy
- scipy
- scikit-learn
- matplotlib
- ...

To install packages:

```
conda install PACKAGE_NAME
```

```
pip install PACKAGE_NAME
```

Let's run our slides first!

```
jupyter notebook
```

Want more fancy stuff?

```
conda install -c damianavila82 rise
```

Play with your toys!

In [1]:
```
print ('Hello Jupyter Notebook!')
```

```
Hello Jupyter Notebook!
```

Python Basics

- Data Types
- Containers
- Functions
- Classes

Basic data types

Numbers

Integers and floats work as you would expect from other languages:

In [2]:
```
x = 3
print (x, type(x))
```

```
3 <class 'int'>
```

In [3]:
```
print (x + 1)   # Addition;
print (x - 1)   # Subtraction;
print (x * 2)   # Multiplication;
print (x ** 2)  # Exponentiation;
```

```
4
2
6
9
```

In [4]:
```
x += 1
print (x)  # Prints "4"
x *= 2
print (x)  # Prints "8"
```

```
4
8
```

In [5]:
```
y = 2.5
print (type(y)) # Prints "<type 'float'>"
print (y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation (https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex).

In [6]:
```
print (17 / 3)   # return float
print (17 // 3)  # return integer
print (17 % 3)   # Modulo operation
```

```
5.666666666666667
5
2
```

Booleans

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):

In [7]:
```
t, f = True, False # Note the Captilzation!
print (type(t)) # Prints "<type 'bool'>"
```

```
<class 'bool'>
```

Now we let's look at the operations:

In [8]:
```
print (t and f) # Logical AND;
print (t or f)  # Logical OR;
print (not t)   # Logical NOT;
print (t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings

In [9]:
```
hello = 'hello'   # String literals can use single quotes
world = "world"   # or double quotes; it does not matter.
print (hello, len(hello))
```

```
hello 5
```

In [10]:
```
hw = hello + ' ' + world  # String concatenation
print (hw)  # prints "hello world"
```

```
hello world
```

In [33]:
```
# sprintf style string formatting
hw12 = '%s %s %d' % (hello, world, 12)
# Recommended formatting style for Py3.0+ (https://pyformat.info)
new_py3_hw12 = '{:>15} {:1.1f} {}'.format('hello' + ' ' +  'world', 1,
2)
print (hw12, new_py3_hw12)
```

```
hello world 12     hello world 1.0 2
```

In [12]:
```
s = "hello"
print (s.capitalize())  # Capitalize a string; prints "Hello"
print (s.upper())       # Convert a string to uppercase; prints "HELLO"
print (s.rjust(7))      # Right-justify a string, padding with spaces; p
rints "  hello"
print (s.center(7))     # Center a string, padding with spaces; prints "
 hello "
print (s.replace('l', '(ell)'))  # Replace all instances of one substrin
g with another;
                                 # prints "he(ell)(ell)o"
print ('  world '.strip())  # Strip leading and trailing whitespace; pri
nts "world"
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

In [13]:
```
"You can type ' inside"
```

Out[13]:
```
"You can type ' inside"
```

In [14]:
```
'You can type " inside'
```

Out[14]:
```
'You can type " inside'
```

You can find a list of all string methods in the document (https://docs.python.org/3/library/string.html).

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists

A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

In [15]:
```
x = [1, 2, 3, 'a', 'b', 'c'] + ['hello'] # list append with the + operat
or
print (x, x[2])  # access by index
print (x[-1])    # index can be negative
```

```
[1, 2, 3, 'a', 'b', 'c', 'hello'] 3
hello
```

In [16]:
```
x.append('element')
print (x)
print (x.pop(), x)
```

```
[1, 2, 3, 'a', 'b', 'c', 'hello', 'element']
element [1, 2, 3, 'a', 'b', 'c', 'hello']
```

Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

In [17]:
```
x = [1, 2, 3, 4, 5]
print (x[2:])
print (x[:-1])
print (x[2:5])
x[0:3] = ['a', 'b', 'c']  # modify elements in list
print (x)
```

```
[3, 4, 5]
[1, 2, 3, 4]
[3, 4, 5]
['a', 'b', 'c', 4, 5]
```

In [18]:
```
y = x[:]   # copy list
y[2] = 100 # x won't change
print ('y:', y)
print ('x:', x)
```

```
y: ['a', 'b', 100, 4, 5]
x: ['a', 'b', 'c', 4, 5]
```

As usual, you can find all the gory details about lists in the [documentation (https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range)](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range).

Loops

You can loop over the elements of a list like this:

In [19]:
```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print (animal)
```

```
cat
dog
monkey
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

In [20]:
```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print ('#%d: %s' % (idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

List comprehensions:

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

In [21]:
```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print (squares)
```

```
[0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

In [22]:
```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print (squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

In [23]:
```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print (even_squares)
```

```
[0, 4, 16]
```

In [24]:
```
nums = [0, 1, 2, 3, 4]
even_squares_or_one = [x ** 2 if x % 2 == 0 else 1 for x in nums]
print (even_squares_or_one)
```

```
[0, 1, 4, 1, 16]
```

Dictionaries

A dictionary stores (key, value) pairs, similar to a Map in C++. You can use it like this:

In [25]:
```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some
 data
print (d['cat'])       # Get an entry from a dictionary; prints "cute"
print ('cat' in d)     # Check if a dictionary has a given key; prints
 "True"
```

```
cute
True
```

In [26]:
```
d['fish'] = 'wet'    # Set an entry in a dictionary
print (d['fish'])      # Prints "wet"
```

```
wet
```

In [27]:
```
print (d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
-----------------------------------------------------------------------
----
KeyError                                  Traceback (most recent call l
ast)
<ipython-input-27-39608aeda0ef> in <module>()
----> 1 print (d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

In [ ]:
```
print (d.get('monkey', 'N/A'))  # Get an element with a default; prints
 "N/A"
print (d.get('fish', 'N/A'))    # Get an element with a default; prints
 "wet"
```

In [ ]:
```
del d['fish']         # Remove an element from a dictionary
print (d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

You can find all you need to know about dictionaries in the documentation (https://docs.python.org/3/library/stdtypes.html#mapping-types-dict).

It is easy to iterate over the keys in a dictionary:

In [ ]:
```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print ('A %s has %d legs' % (animal, legs))
```

If you want access to keys and their corresponding values, use the items method:

In [ ]:
```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print ('A %s has %d legs' % (animal, legs))
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print (even_num_to_square)
```

In [ ]:
```
# Make a dictionary from two lists using zip
l1 = ['EECS445', 'EECS545']
l2 = ['Undergraduate ML', 'Graduate ML']
d = zip(l1, l2)
# Unroll dictionary into two lists
l3, l4 = zip(*d)
print
```

Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

In [ ]:
```
animals = {'cat', 'dog'}
print ('cat' in animals)   # Check if an element is in a set; prints "Tr
ue"
print ('fish' in animals)  # prints "False"
```

In [ ]:
```
animals.add('fish')      # Add an element to a set
print ('fish' in animals)
print (len(animals))       # Number of elements in a set;
```

In [ ]:
```
animals.add('cat')       # Adding an element that is already in the set
 does nothing
print (len(animals))
animals.remove('cat')    # Remove an element from a set
print (len(animals))
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

In [ ]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print ('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

In [ ]:
```
from math import sqrt
print ({int(sqrt(x)) for x in range(30)})
```

Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

In [ ]:
```
d = {(x, x + 1): x for x in range(0, 10, 2)}  # Create a dictionary with
 tuple keys, note that range can use step args.
t = (0, 1)       # Create a tuple
print (type(t))
print (d[t])
print (d[(2, 3)])
```

In [ ]:
```
t[0] = 1
```

Functions

Python functions are defined using the def keyword. For example:

In [ ]:
```
def get_GPA(x):
    if x >= 90:
        return "A"
    elif x >= 75:
        return "B"
    elif x >=60:
        return "C"
    else:
        return "F"

for x in [59, 70, 91]:
    print (get_GPA(x))
```

We will often define functions to take optional keyword arguments, like this:

In [ ]:
```
def fib(n = 10):
    a = 0
    b = 1
    while b < n:
        print(b, end=',')
        a, b = b, a + b

fib()
```

Classes

The syntax for defining classes in Python is straightforward:

In [ ]:
```
class Greeter:

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print ('HELLO, %s!' % self.name.upper())
        else:
            print ('Hello, %s' % self.name)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

Modules

- import modules
- numpy
- matplotlib
- scikit-learn

In [ ]:
```
from modules import fibo
fibo.fib2(10)
```

NumPy

- NumPy arrays, dtype, and shape
- Reshape and Update In-Place
- Combine Arrays
- Array Math
- Inner Product
- Matrixes

To use Numpy, we first need to import the `numpy` package:

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

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

In [ ]:
```
b = np.array([[0, 2, 4], [1, 3, 5]], dtype = np.float64)
print(b)
print(b.shape)
print(b.dtype)
```

Numpy also provides many functions to create arrays:

In [ ]:
```
np.zeros(5)  # Create an array of all zeros
```

In [ ]:
```
np.ones(shape=(3, 4), dtype = np.int32)  # Create an array of all ones
```

In [ ]:
```
np.full((2,2), 7, dtype = np.int32)  # Create a constant array
```

In [ ]:
```
np.eye(2)  # Create a 2x2 identity matrix
```

```
In [ ]: np.random.random((2,2))  # Create an array filled with random values
```

Array indexing

Numpy offers several ways to index into arrays. Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
In [ ]: # Create the following rank 2 array with shape (3, 4)
        # [[ 1  2  3  4]
        #  [ 5  6  7  8]
        #  [ 9 10 11 12]]
        a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

        # Use slicing to pull out the subarray consisting of the first 2 rows
        # and columns 1 and 2; b is the following array of shape (2, 2):
        # [[2 3]
        #  [6 7]]
        b = a[:2, 1:3]
        print (b)
```

```
In [ ]: print (a[0, 1])
        b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
        print (a[0, 1])
```

```
In [ ]: row_r1 = a[1, :]    # Rank 1 view of the second row of a
        row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
        row_r3 = a[[1], :]  # Rank 2 view of the second row of a
        print (row_r1, row_r1.shape)
        print (row_r2, row_r2.shape)
        print (row_r3, row_r3.shape)
```

Reshape and Update In-Place

```
In [ ]: e = np.arange(12)
        print(e)
```

```
In [ ]: # f is a view of contents of e
        f = e.reshape(3, 4)
        print(f)
```

```
In [ ]: # Set values of e from index 5 onwards to 0
        e[7:] = 0
        print (e)
        # f is also updated
        print (f)
```

```
In [ ]: # We can get transpose of array by T attribute
        print (f.T)
```

Combine Arrays

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

In [ ]:
```
b = np.array([[1, 2, 3], [4, 5, 6]])
d = b / 2.0
# Use broadcasting when needed to do this automatically
print (np.vstack([a, b, d]))
```

In [ ]:
```
# In machine learning, useful to enrich or
# add new/concatenate features with hstack
np.hstack([b, d])
```

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

In [ ]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
```

In [ ]:
```
# Elementwise sum; both produce the array
print (x + y)
print (np.add(x, y))
```

In [ ]:
```
# Elementwise difference; both produce the array
print (x - y)
print (np.subtract(x, y))
```

In [ ]:
```
# Elementwise product; both produce the array
print (x * y)
print (np.multiply(x, y))
```

In [ ]:
```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print (x / y)
print (np.divide(x, y))
```

In [ ]:
```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print (np.sqrt(x))
```

Broadcasting

Arrays with different dimensions can also perform above operations.

In [ ]:
```
# Multiply single number
print (x * 0.5)
```

In [ ]:
```
a = np.array([1, 2, 3])
b = np.array([[1, 2, 3], [4, 5, 6]])
```

In [ ]:
```
c = a + b
print(a.shape, b.shape, c.shape)
print(c)
```

In [ ]:
```
a.reshape((1, 1, 3)) + c.reshape((2, 1, 3))
```

We can also get statistical results directly using `sum`, `mean` and `std` methods.

In [ ]:
```
print (d.sum())
print (d.sum(axis = 0))
print (d.mean())
print (d.mean(axis = 1))
print (d.std())
print (d.std(axis = 0))
```

Inner Product

$$(a_1, a_2, a_3, \ldots, a_n) \cdot (b_1, b_2, b_3, \ldots, b_n)^T = \sum_{i=1}^{n} a_i b_i$$

We use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

In [ ]:
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print (v.dot(w))
print (np.dot(v, w))
```

In [ ]:
```
# Matrix / vector product; both produce the rank 1 array [29 67]
print (x.dot(v))
print (np.dot(x, v))
```

In [ ]:

```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print (x.dot(y))
print (np.dot(x, y))
```

Matrix

Instead of arrays, we can also use matrix to simplify the code.

In [ ]:

```
x = np.matrix('1, 2, 3; 4, 5, 6')
y = np.matrix(np.ones((3, 4)))
print(x.shape)
print(y.shape)
print(x * y)
print(y.T * x.T)
```

You can find more in http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html (http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html).

Matplotlib

- Plotting Lines
- Plotting Multiple Lines
- Scatter Plots
- Legend, Titles, etc.
- Subplots
- Histogram

In [ ]:

```
import pylab as plt
```

To make pylab work inside ipython:

In [ ]:

```
%matplotlib inline
```

In [ ]:

```
plt.plot([1,2,3,4], 'o-')
plt.ylabel('some numbers')
plt.show()
```

In [ ]:

```
x = np.linspace(0,1,100);
y1 = x ** 2;
y2 = np.sin(x);

plt.plot(x, y1, 'r-', label="parabola");
plt.plot(x, y2, 'g-', label="sine");
plt.legend();
plt.xlabel("x axis");
plt.show()
```

In [ ]:

```
# Create sample data, add some noise
x = np.random.uniform(1, 100, 1000)
y = np.log(x) + np.random.normal(0, .3, 1000)

plt.scatter(x, y)
plt.show()
```

Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

In [ ]:

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# First plot
plt.subplot(2, 1, 1)
plt.plot(x, y_sin)
plt.title('Sine')

# Second plot
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

In [ ]:
```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Scikit-learn

This is a common machine learning package with lots of algorithms, you can find detailed usage [here (http://scikit-learn.org/stable/documentation.html)](http://scikit-learn.org/stable/documentation.html).

Here is an example of KMeans cluster algorithm:

In [ ]:
```
from sklearn.cluster import KMeans
```

In [ ]:
```
mu1 = [5, 5]
mu2 = [0, 0]
cov1 = [[1, 0], [0, 1]]
cov2 = [[2, 1], [1, 3]]
x1 = np.random.multivariate_normal(mu1, cov1, 1000)
x2 = np.random.multivariate_normal(mu2, cov2, 1000)

print (x1.shape)
print (x2.shape)

plt.plot(x1[:, 0], x1[:, 1], 'r.')
plt.plot(x2[:, 0], x2[:, 1], 'b.')
plt.show()
```

In [ ]:
```
x = np.vstack([x1, x2])
print (x.shape)
plt.plot(x[:, 0], x[:, 1], 'b.')
plt.show()
```

In [ ]:

```
y_pred = KMeans(n_clusters=2).fit_predict(x)
x_pred1 = x[y_pred == 0, :]
x_pred2 = x[y_pred == 1, :]
print (x_pred1.shape)
print (x_pred2.shape)
plt.plot(x_pred1[:, 0], x_pred1[:, 1], 'b.')
plt.plot(x_pred2[:, 0], x_pred2[:, 1], 'r.')
plt.show()
```