CSC411 - Python Tutorial

Kaustav Kundu

October 5, 2016

Why Python?

1. High level scripting language.
2. FOSS (Free and Open Source Software) - unlike Matlab.
3. Extremely good documentation (https://www.python.org/doc/) and support (Stack Overflow, etc.).
4. Rich library of modules, including third party modules/add-ons. Eg. numpy, scipy, pandas, matplotlib, scikit-learn, Pylearn2, tensorflow, caffe, theano, etc.

Operations

1. Arithmetic
   $+, -, *, /, \%(\text{modulus}), **(\text{exponent}), //(\text{floor division})$
2. Relational
   $>, <, >=, <=, ==, !=$
3. Logical
   `or`, `and`, `not`
4. Bitwise
   &, $|$, $\wedge$, $\sim$, $<<, >>$

Data Structures: Lists

1. Initialization

```
In [1]: a = [1, 2, 3, 4, 5]

In [2]: range(1, 6)
Out[2]: [1, 2, 3, 4, 5]

In [3]: range(1, 6, 2)
Out[3]: [1, 3, 5]

In [4]: [1] * 5
Out[4]: [1, 1, 1, 1, 1]
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

```
In [9]: a = range(1, 6)
        b = range(1, 6, 2)

In [10]: a.append(b)

In [11]: a

Out[11]: [1, 2, 3, 4, 5, [1, 3, 5]]
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

```
In [12]: a = range(1, 6)
         b = range(1, 6, 2)

In [13]: a.extend(b)

In [14]: a

Out[14]: [1, 2, 3, 4, 5, 1, 3, 5]
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

```
In [15]: a.insert(2, 10)

In [16]: a
Out[16]: [1, 2, 10, 3, 4, 5, 1, 3, 5]
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

```
In [17]: a.count(1)

Out[17]: 2
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

```
In [18]: a.index(5)

Out[18]: 5
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

```
In [22]: a

Out[22]: [1, 1, 2, 3, 3, 4, 5, 5, 10]
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`

In [26]:
```
a = [5, 3, 1, 5, 4, 3, 10, 2, 1]
a.reverse()
print(a)
```

```
[1, 2, 10, 3, 4, 5, 1, 3, 5]
```

In [28]:
```
a = [5, 3, 1, 5, 4, 3, 10, 2, 1]
print(a[::-1])
print(a)
```

```
[1, 2, 10, 3, 4, 5, 1, 3, 5]
[5, 3, 1, 5, 4, 3, 10, 2, 1]
```

Data Structures: Lists

1. Initialization
2. Methods: `append`, `extend`, `insert`, `remove`, `count`, `index`, `sort`, `reverse`
3. List comprehensions

```
In [29]: [x-y for x in range(0, 3) for y in range(0, 5)]
Out[29]: [0, -1, -2, -3, -4, 1, 0, -1, -2, -3, 2, 1, 0, -1, -2]
```

Data Structures: Lists

1. "Sum" of lists

```
In [47]: a
Out[47]: [5, 3, 1, 5, 4, 3, 10, 2, 1]

In [48]: b = a[::-1]

In [49]: b
Out[49]: [1, 2, 10, 3, 4, 5, 1, 3, 5]

In [50]: a + b
Out[50]: [5, 3, 1, 5, 4, 3, 10, 2, 1, 1, 2, 10, 3, 4, 5, 1, 3, 5]
```

Data Structures: Lists

1. "Sum" of lists

```
In [47]: a
Out[47]: [5, 3, 1, 5, 4, 3, 10, 2, 1]

In [48]: b = a[::-1]

In [49]: b
Out[49]: [1, 2, 10, 3, 4, 5, 1, 3, 5]

In [50]: a + b
Out[50]: [5, 3, 1, 5, 4, 3, 10, 2, 1, 1, 2, 10, 3, 4, 5, 1, 3, 5]

In [52]: [x + y for x, y in zip(a, b)]
Out[52]: [6, 5, 11, 8, 8, 8, 11, 5, 6]
```

Data Structures: Lists

1. "Sum" of lists

2. 2D lists

```
In [53]: c = [[x for x in range(0, y)] for y in range(0, 5)]
         c
Out[53]: [[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3]]

In [54]: c[3][1], c[3][2]
Out[54]: (1, 2)

In [55]: len(c[3]), len(c)
Out[55]: (3, 5)
```

Data Structures: Arrays

- Initialization

```
In [56]: import numpy as np

In [57]: np.array([1, 2, 3])
Out[57]: array([1, 2, 3])

In [58]: np.array([[1, 2], [3, 4]])
Out[58]: array([[1, 2],
                [3, 4]])

In [59]: np.ones((5,), dtype=np.int)
Out[59]: array([1, 1, 1, 1, 1])

In [60]: np.zeros(5)
Out[60]: array([ 0.,  0.,  0.,  0.,  0.])

In [61]: np.arange(0, 6)
Out[61]: array([0, 1, 2, 3, 4, 5])
```

Data Structures: Arrays

- Initialization
- Element-wise operations

```
In [63]: a = np.arange(1, 6)
         a
Out[63]: array([1, 2, 3, 4, 5])

In [64]: a[1:3]
Out[64]: array([2, 3])

In [66]: b = np.random.randint(1, 100, 5)
         b
Out[66]: array([65, 64,  1, 10, 10])
```

Data Structures: Arrays

- Initialization
- Element-wise operations

```
In [67]: a + b
Out[67]: array([66, 66,  4, 14, 15])

In [68]: a / b
Out[68]: array([0, 0, 3, 0, 0])

In [69]: a*1.0 / b
Out[69]: array([ 0.01538462,  0.03125   ,  3.        ,  0.4
          ,  0.5       ])

In [70]: a.astype(np.float)/b
Out[70]: array([ 0.01538462,  0.03125   ,  3.        ,  0.4
          ,  0.5       ])
```

Data Structures: Arrays

- Dot Products

```
In [72]: c = np.random.randint(10, size=(3,3))
         c
```

```
Out[72]: array([[0, 2, 6],
                [8, 8, 6],
                [7, 7, 0]])
```

```
In [74]: b = np.ones((3,1)) * 2
         b
```

```
Out[74]: array([[ 2.],
                [ 2.],
                [ 2.]])
```

```
In [75]: np.dot(c, b)
```

```
Out[75]: array([[ 16.],
                [ 44.],
                [ 28.]])
```

```
In [76]: c.dot(b)
```

```
Out[76]: array([[ 16.],
                [ 44.],
                [ 28.]])
```

Data Structures: Arrays

- Matrix Multiplication

```
In [77]: b = np.ones((3, 3))
         c = np.eye(3)
         b * c

Out[77]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In [78]: b.dot(c)

Out[78]: array([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])
```

Data Structures: Arrays

- Matrix Multiplication
- Transpose

```
In [80]: c

Out[80]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In [81]: c.T

Out[81]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

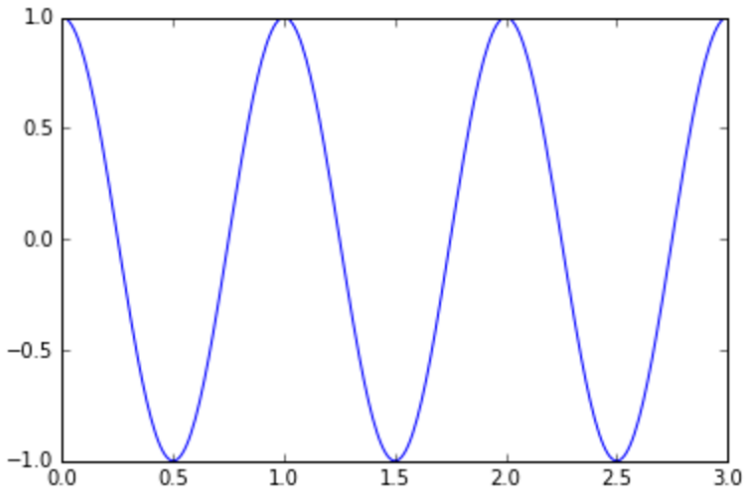
Data Structures: Arrays vs Lists

1. Difference between lists and arrays is similar to that between cell arrays and matrices in MATLAB.
2. All elements in arrays have to be of the same data type, specified at the time of creation.
3. Arrays are more memory efficient than lists because of well-defined data type.

Data Structures: Arrays vs Lists

```
In [86]: import matplotlib.pyplot as plot
         t = np.arange(0, 3, 0.01)
         F = 2
         x = np.cos(np.math.pi*t*F)
         plot.plot(t, x)
```

```
Out[86]: [<matplotlib.lines.Line2D at 0x10cbde050>]
```



1. `numpy.array` provides a lot of advantages to perform matrix operations, like transpose, inverse, eigen values, etc.
2. For more details look at `numpy` and `scipy` documentations.

Data Structures: Dictionary

Equivalent to `std::map` in C++.

```
In [88]: d = {str(x): x**2 for x in range(0, 5)}
         d
Out[88]: {'0': 0, '1': 1, '2': 4, '3': 9, '4': 16}

In [89]: d['4']
Out[89]: 16
```

Modules

```
# Fibonacci numbers module

def fib(n):   # write Fibonacci series up to n
   a, b = 0, 1
   while b < n:
      print b,
      a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
   result = []
   a, b = 0, 1
   while b < n:
      result.append(b)
      a, b = b, a+b
   return result
```

```
In [90]: import fibo

In [91]: fibo.fib(1000)
         1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

In [93]: f = fibo.fib2(1000)
         f
Out[93]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]

In [94]: fibo.__name__
Out[94]: 'fibo'
```

Mutable and Immutable objects

1. (Almost) everything in python is an object.
2. Rather than focusing on pass by value or pass by reference, we should think whether an object is mutable or immutable.

In [103]:
```
some_name = 'CSC'

course_prefix = []
course_prefix.append(some_name)

course_name = course_prefix
course_name.append('411')
some_name = 'STA'

print (some_name, course_prefix, course_name)
```

```
('STA', ['CSC', '411'], ['CSC', '411'])
```

1. Changes to `course_name`/`course_prefix` does not affect `some_name`, since `some_name` immutable.
2. Any change to `course_name`, affects `course_prefix`, since `course_prefix` is mutable.

Mutable and Immutable objects: Examples

```
In [105]: a = (some_name, course_prefix, course_name)
          print(a)
          course_prefix.append('H1')
          print(a)
```

```
('STA', ['CSC', '411', 'H1'], ['CSC', '411', 'H1'])
('STA', ['CSC', '411', 'H1', 'H1'], ['CSC', '411', 'H1', 'H1'])
```

1. Though tuples are immutable, but it contains references to mutable objects.
2. Function calls work similarly.

Mutable and Immutable objects: Examples

```
In [105]: a = (some_name, course_prefix, course_name)
          print(a)
          course_prefix.append('H1')
          print(a)
```

```
('STA', ['CSC', '411', 'H1'], ['CSC', '411', 'H1'])
('STA', ['CSC', '411', 'H1', 'H1'], ['CSC', '411', 'H1', 'H1'])
```

1. Though tuples are immutable, but it contains references to mutable objects.

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [106]:
```
def foo(a):
    a = 'another string'
    print(a)

val = 'string'
foo(val)
print(val)
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [106]:
```
def foo(a):
    a = 'another string'
    print(a)

val = 'string'
foo(val)
print(val)
```

```
another string
string
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [106]:
```
def foo(a):
    a = 'another string'
    print(a)

val = 'string'
foo(val)
print(val)
```

```
another string
string
```

In [107]:
```
def foo(a):
    a.append(5)
    print(a)
a = [10]
foo(a)
print(a)
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [106]:
```
def foo(a):
    a = 'another string'
    print(a)

val = 'string'
foo(val)
print(val)
```

```
another string
string
```

In [107]:
```
def foo(a):
    a.append(5)
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 5]
[10, 5]
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [107]:
```
def foo(a):
    a.append(5)
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 5]
[10, 5]
```

In [109]:
```
def foo(a):
    a = a + a
    print(a)
a = [10]
foo(a)
print(a)
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [107]:
```
def foo(a):
    a.append(5)
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 5]
[10, 5]
```

In [109]:
```
def foo(a):
    a = a + a
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 10]
[10]
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [107]:
```
def foo(a):
    a.append(5)
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 5]
[10, 5]
```

In [110]:
```
def foo(a):
    a += a
    print(a)
a = [10]
foo(a)
print(a)
```

Mutable and Immutable objects: Examples

1. Function calls work similarly.

In [107]:
```
def foo(a):
    a.append(5)
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 5]
[10, 5]
```

In [110]:
```
def foo(a):
    a += a
    print(a)
a = [10]
foo(a)
print(a)
```

```
[10, 10]
[10, 10]
```

Mutable and Immutable objects: Examples

Immutable

1. Numeric types: `int`, `float`, `complex`
2. `string`
3. `tuple`
4. `frozen set`
5. `bytes`

Mutable

1. `list`
2. `dict`
3. `set`
4. `byte array`

Classes

In [95]:
```
class Complex(object):
    def __init__(self, real_part, imag_part):
        self.r = real_part
        self.im = imag_part
x = Complex(3.0, -4.0)
x.r, x.im
```

Out[95]:
```
(3.0, -4.0)
```

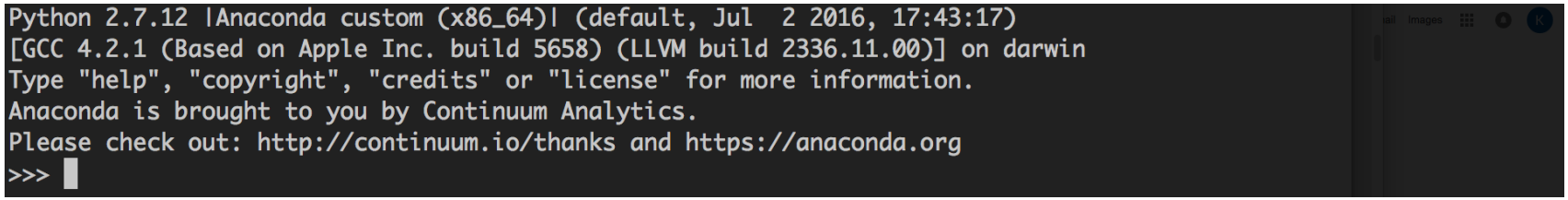
In [96]:
```
type(x)
```

Out[96]:
```
__main__.Complex
```

Python Development Environment

1. **Python shell**: REPL environment

   Type `python` from the command line to use the python interpreter



Python Development Environment

1. **Python shell**: REPL environment
2. **iPython**: Advanced interactive python shell
   1. Tab completion
   2. Better history management
   3. More explicit and color highlighted error messages
   4. Besides the shell interface, it also has a web interface with support for code, inline plots, etc and can be converted to pdf, html, latex documents.

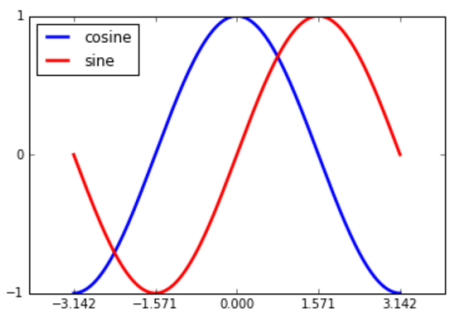
```
In [1]: import numpy as np
        import matplotlib.pyplot as plot

In [2]: X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
        C, S = np.cos(X), np.sin(X)

In [3]: %matplotlib inline

In [4]: plot.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
        plot.plot(X, S, color="red",  linewidth=2.5, linestyle="-", label="sine")
        plot.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
        plot.yticks([-1, 0, +1])
        plot.legend(loc='upper left')

Out[4]: <matplotlib.legend.Legend at 0x10f1b44d0>
```



Python Development Environment

1. **Python shell**: REPL environment
2. **iPython**: Advanced interactive python shell
   1. Tab completion
   2. Better history management
   3. More explicit and color highlighted error messages
   4. Besides the shell interface, it also has a web interface with support for code, inline plots, etc and can be converted to pdf, html, latex documents.
3. **Editor**
   (a) Text Editors: vim, emacs, Sublime Text
   (b) IDE: PyCharm, Spyder, Visual Studio

Python Environment

Package Manager

1. Python Packaging Authority (PyPA): integrated with default python installation
    1. pip
    2. setuptools
2. Non PyPA
    1. Anaconda

Virtual environments (Isolating python packages among multiple projects)

1. PyPA
    1. virtualenv (Python 2.6+ and Python3.3+)
    2. venv (Python 3.3+)
2. Non PyPA
    1. Anaconda

Help

Tutorials:

1. Tutorials Point
2. Learn X in Y minutes

References

1. Stack Overflow
2. Numpy for Matlab users: [Link 1, Link 2]

Assignment 1: Displaying digits

```
In [1]: from utils import *
        from plot_digits import *

In [2]: input, target = load_train()

In [3]: %matplotlib inline

In [4]: plot_digits(input)
```

```
Displaying pane 2/20
```