Python Basics

June 20th 2018

Xiangyu Qu

Use package manager (pip)

Show version and root of pip	pip --version **or** pip3 --version
List all installed packages	pip3 list
Show information about a specific installed package	pip3 show [package-name] (e.g. pip3 show numpy)
Search for a specific package in PyPI repository	pip3 search [package-name]
Install packages	pip3 install [package-name 1] [package-name 2]
Uninstall packages	pip3 uninstall [package-name 1] [package-name 2]
Upgrade a package	pip3 install --upgrade [package-name]
Install a specific version of a package	pip3 install [package-name]==version (e.g. pip3 install numpy==1.14.3)
Check dependencies	pip3 check
Most important one	pip3 --help

Python virtual environment

Create a virtual environment

in python3.4 and above: python3 –m venv [some optional arguments] path/to/environment/directory

in python3.3 and python2: virtualenv [some optional arguments] path/to/environment/directory

example: python3 –m venv --system-site-packages ./workshop

Activate a virtual environment

bash: source <virtual environment directory>/bin/activate

cmd.exe: <virtual environment directory>\Scripts\activate.bat

Exit a virtual environment

deactivate

Get help

python3 –m venv -h

Numpy

- Numpy is a scientific computation library for Python

- It is very similar to MATLAB. For a side-by-side comparison, visit http://scipy.github.io/old-wiki/pages/NumPy_for_Matlab_Users

- The basic unit for computation in Numpy is ndarray, which is similar to array in MATLAB. Numpy ndarray is 0 indexed, MATLAB array is 1 indexed

Numpy

- Create array from a list of numbers and element access

Numpy

```
In[2]: import numpy as np
In[3]: a = np.array([1, 2, 3])
In[4]: print(a)
[1 2 3]
In[5]: a = np.array([[1, 2], [3, 4]])
In[6]: print(a)
[[1 2]
 [3 4]]
In[7]: print('rank:',a.ndim,'shape:',a.shape,'number of elements:',a.size)
rank: 2 shape: (2, 2) number of elements: 4
In[8]: print('a[0] =',a[0,0])
a[0] = 1
```

MATLAB

```
>> a = [1, 2 ,3]
a =
     1     2     3
>> a = [1, 2; 3, 4]
a =
     1     2
     3     4
>> disp(ndims(a))
     2
>> disp(size(a))
     2     2
>> disp(numel(a))
     4
>> disp(a(1,1))
     1
```

Numpy

- Create array from built-in functions

Numpy

```
In[17]:
In[17]: a = np.ones((2,2))
In[18]: print(a)
[[1. 1.]
 [1. 1.]]
In[19]: print(np.zeros((2, 2)))
[[0. 0.]
 [0. 0.]]
In[20]: print(np.full((2, 2), 0.5))
[[0.5 0.5]
 [0.5 0.5]]
In[21]: print(np.eye(2))
[[1. 0.]
 [0. 1.]]
In[22]: print(np.random.rand(2, 2))
[[0.7881301  0.14243331]
 [0.63433755 0.03357696]]
```

MATLAB

```
>> ones(2,2)
ans =
     1     1
     1     1
>> disp(zeros(2,2))
     0     0
     0     0
>> disp(eye(2))
     1     0
     0     1
>> disp(rand(2))
    0.8147    0.1270
    0.9058    0.9134
>>
```

Numpy

- Array indexing: slicing; syntax: b = a[i:j] ::== return all elements between i and j-1

Numpy

```
In[2]: import numpy as np
In[3]: a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
In[4]: print(a)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
In[5]: b = a[1:3, 1:]
In[6]: print(b)
[[ 6  7  8]
 [10 11 12]]
In[7]: print(a[0:1, :])
[[1 2 3 4]]
In[8]: print(a[1:2, :-2])
[[5 6]]
In[9]: print('integer row index:',a[0,:].ndim)
integer row index: 1
In[10]: print('slice row index:',a[0:1,:].ndim)
slice row index: 2
```

MATLAB

```
>> a = [1,2,3,4; 5,6,7,8; 9,10,11,12]
a =
     1     2     3     4
     5     6     7     8
     9    10    11    12
>> b = a(2:3, 2:end)
b =
     6     7     8
    10    11    12
>> a(1,:)
ans =
     1     2     3     4
>> a(-1, -1)
```

Numpy

- Array indexing: get a slice with another integer array or Boolean array

```
In[11]: print(a[[0,1,2], [2,1,0]])
[3 6 9]
In[12]: print(np.array([a[0,2], a[1,1], a[2,0]]))
[3 6 9]
In[13]: b = np.array([0,1,2])
In[14]: print(a[b, b[::-1]])
[3 6 9]
In[15]: print(a)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
In[16]: a[b, b[::-1]] += 10
In[17]: print(a)
[[ 1  2 13  4]
 [ 5 16  7  8]
 [19 10 11 12]]
```

```
In[20]: print(a)
[[ 1  2 13  4]
 [ 5 16  7  8]
 [19 10 11 12]]
In[21]: print(a[a > 6])
[13 16  7  8 19 10 11 12]
In[22]: a[a>6] = -a[a>6]
In[23]: print(a)
[[  1   2 -13   4]
 [  5 -16  -7  -8]
 [-19 -10 -11 -12]]
```

Numpy

- **Note: in Numpy, a slicing operation does not create a new copy of the original data but another view of the data; changes on slice will change the corresponding values in the original array, and vice versa**

```
In[23]: a_array = np.random.rand(3,3)
In[24]: print(a_array)
[[0.18939277 0.66030958 0.42553664]
 [0.27264523 0.24132075 0.38122089]
 [0.40534569 0.64090631 0.35929866]]
In[25]: a_slice = a_array[:2,:2]
In[26]: print(a_slice)
[[0.18939277 0.66030958]
 [0.27264523 0.24132075]]
In[27]: a_slice[1,1] = 0
In[28]: print(a_array)
[[0.18939277 0.66030958 0.42553664]
 [0.27264523 0.         0.38122089]
 [0.40534569 0.64090631 0.35929866]]
In[29]: a_array[0, 0] = -1
In[30]: print(a_slice)
[[-1.         0.66030958]
 [ 0.27264523  0.        ]]
```

Numpy

- Array math: default Numpy array arithmetics are elementwise, where in MATLAB defaults are linear algebra

Numpy

```
In[11]: print(a.dot(b))
[[1.5 1.5]
 [3.5 3.5]]
In[12]: np.linalg.inv(a)
Out[12]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]], dtype=float32)
In[13]: np.tile(a, (3,2))
Out[13]:
array([[1., 2., 1., 2.],
       [3., 4., 3., 4.],
       [1., 2., 1., 2.],
       [3., 4., 3., 4.],
       [1., 2., 1., 2.],
       [3., 4., 3., 4.]], dtype=float32)
```

MATLAB

```
>> a * b                                        e(a));
ans =
    1.5000    1.5000
    3.5000    3.5000
>> inv(a)
ans =
   -2.0000    1.0000
    1.5000   -0.5000
>> repmat(a,3,2)
ans =
     1     2     1     2
     3     4     3     4
     1     2     1     2
     3     4     3     4
     1     2     1     2
     3     4     3     4
```

Numpy

- Consider following code:

```
In[35]: a = np.ones((3,3))
In[36]: b = np.ones((3))

In[37]: a + b = ?
```

- will it be

```
array([[2., 2., 2.],
       [2., 2., 2.],
       [2., 2., 2.]])
```

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

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

Numpy

- Broadcasting
  1. Two arrays are considered compatible if they have the same size in corresponding dimensions, or, if one array has a size of 1 in that dimension.
  2. If one array has lower rank than the other one, prepend the lower rank array with 1's until two arrays are of same rank
  3. Given that two arrays are compatible but have different shape, during computation, start from trailing dimension, the array with dimension size 1 is copied along that dimension so that the dimension size matches the other.

Numpy

- Broadcasting

```
In[48]: a = np.ones((3,1,3,))
In[49]: b = np.full((3,1),2,dtype='float32')
In[50]: a.ndim
Out[50]: 3
In[51]: b.ndim
Out[51]: 2
```

1. Prepend dimension of b with 1's so that b.shape becomes (1, 3, 1)
2. Copy b along third dimension. After copying, b.shape becomes (1, 3, 3)
3. Copy a along second dimension. After copying, a.shape becomes (3, 3, 3)
4. Copy b along first dimension. After copying, b.shape becomes (3, 3, 3)

Exercise

Generate a random positive semi-definite 5x5 2D array

Hints:

1. Create a symmetric matrix by adding a random matrix with its transpose
2. Apply eigen decomposition to the matrix; (use np.linalg.eig)
3. Set all negative eigen values to 0; (use Boolean indexing)
4. Compute $Q\Lambda Q^T$; (eigenvalues are stored in a rank 1 array, check out np.diag() function to construct a diagonal matrix from it)

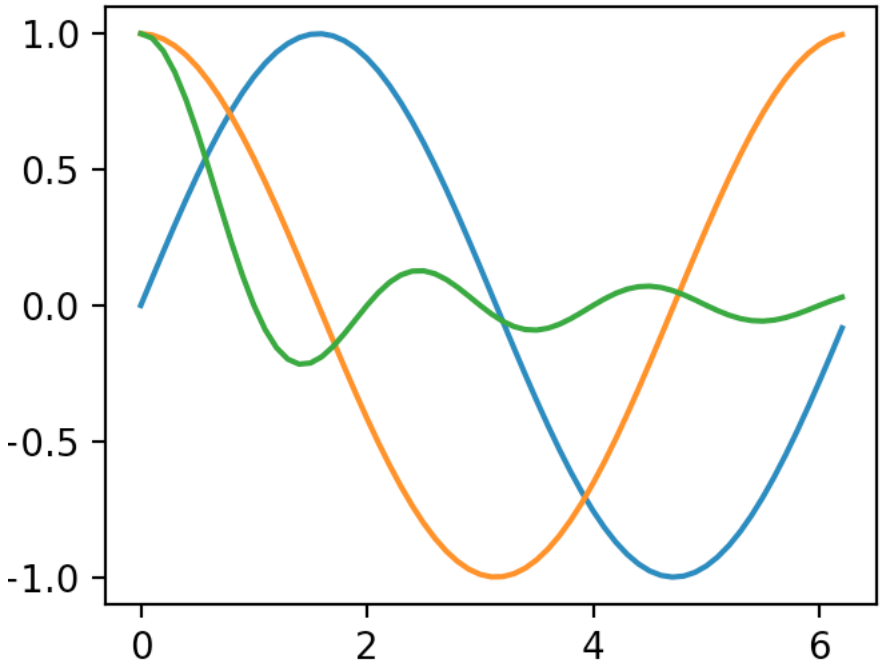
```
import numpy as np
import matplotlib.pyplot as plt

# code for creating a random positive semi-definite matrix
N = 2 # shape of matrix will be NxN
A  = np.random.randn(N, N) # generate a random NxN matrix by drawing from standard normal distribution
A = A + A.transpose() # construct a symmetric matrix
D, V = np.linalg.eig(A) # apply eigen-decomposition to matrix A; D is a 1D array storing eigenvalues, V is a
    2D array
# ...storing eigenvectors
D[D < 0] = 0 # send all negative eigenvalues to 0
D = np.diag(D, 0) # construct a diagonal matrix from eigenvalues stored in D
A_positive_semi = V.dot(D).dot(V.transpose()) # construct positive semi-definite matrix QEQ.'
# line 12 is equivalent to A_positive_semi = np.dot( np.dot(V, D), V.transpose() )
```

Matplotlib

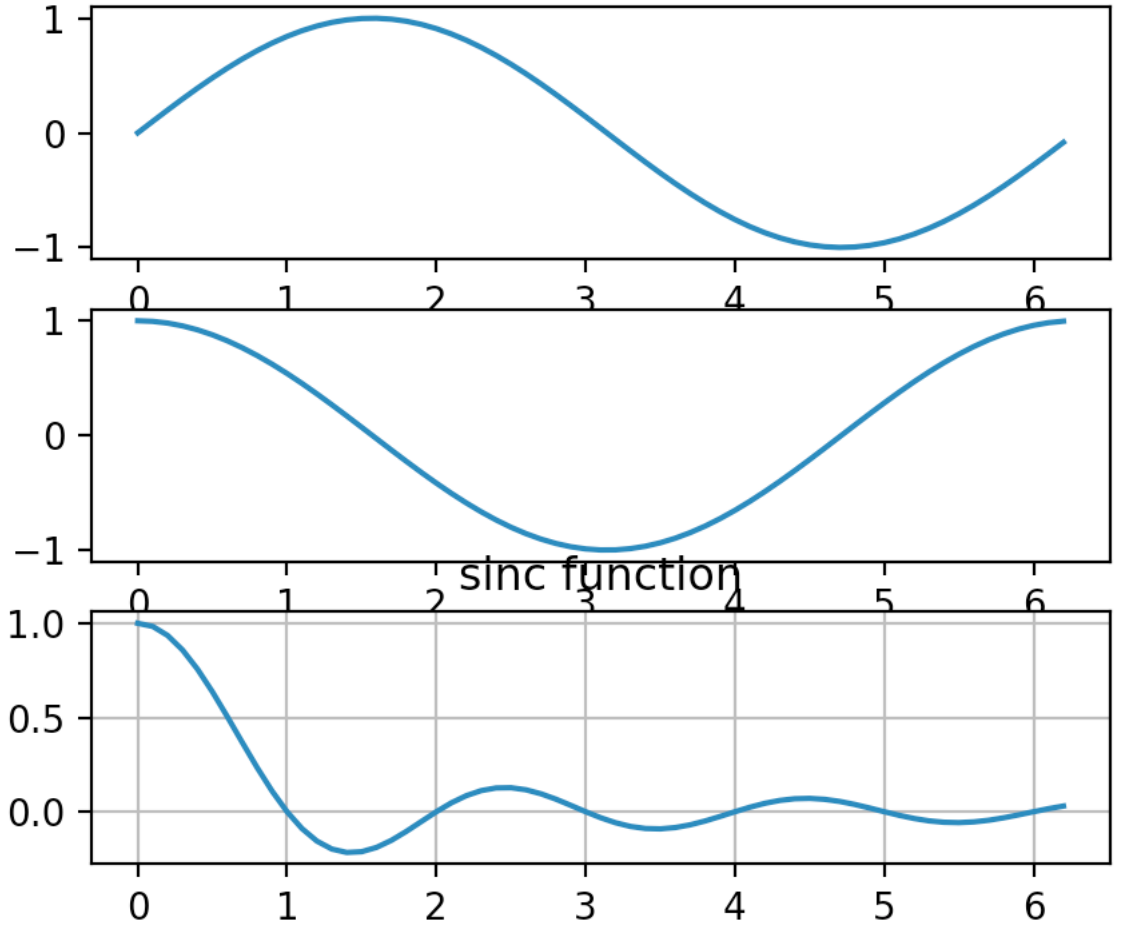
- Matplotlib is a plotting library for Numpy. It can be used to display images, but we will focus on plotting functions

```
In[2]: import numpy as np
In[3]: import matplotlib.pyplot as plt
Backend MacOSX is interactive backend. Turning interactive mode on.
In[4]: x = np.arange(0, 2*np.pi, 0.1)
In[5]: fx = np.sin(x)
In[6]: gx = np.cos(x)
In[7]: hx = np.sinc(x)
In[8]: plt.plot(x, fx)
Out[8]: [<matplotlib.lines.Line2D at 0x111f09c88>]
In[9]: plt.plot(x, gx)
Out[9]: [<matplotlib.lines.Line2D at 0x114cb1438>]
In[10]: plt.plot(x, hx)
Out[10]: [<matplotlib.lines.Line2D at 0x114cbc518>]
```



Matplotlib

```
In[11]: plt.subplot(3,1,1)
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0
In[12]: plt.plot(x, fx)
Out[12]: [<matplotlib.lines.Line2D at 0x114037358>]
In[13]: plt.subplot(3,1,2)
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0
In[14]: plt.plot(x, gx)
Out[14]: [<matplotlib.lines.Line2D at 0x1140792e8>]
In[15]: plt.subplot(3,1,3)
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0
In[16]: plt.plot(x, hx)
Out[16]: [<matplotlib.lines.Line2D at 0x1140de208>]
In[17]: plt.grid
Out[17]: <function matplotlib.pyplot.grid(b=None, wh
In[18]: plt.grid()
In[19]: plt.title('sinc function')
Out[19]: Text(0.5,1,'sinc function')
```



Functions

Python functions are defined using keyword **def**

```
In[19]: def ReLU(x):
   ...:     return x * (x > 0)
   ...:
In[20]: ReLU(-1)
Out[20]: 0
In[21]: ReLU(1.5)
Out[21]: 1.5
```

Function can take keyword arguments and can return multiple outputs

```
In[42]: def ReLU(x, print_result=False, get_grad=False):
   ...:     grad = None
   ...:     if type(x) == list:
   ...:         out = [ele * (ele > 0) for ele in x]
   ...:         if get_grad == True:
   ...:             grad = [int(ele > 0) for ele in x]
   ...:     else:
   ...:         out = x * (x > 0)
   ...:         if get_grad == True:
   ...:             grad = int(x>0)
   ...:     if print_result == True:
   ...:         print('output:',out)
   ...:         print('gradient:',grad)
   ...:     return out, grad
```

```
In[43]: y, y_grad = ReLU(10, True)
output: 10
gradient: None
In[44]: y, y_grad = ReLU([-1.5, -3, 10], get_grad=True)
In[45]: print('output:',y,'\ngradient:',y_grad)
output: [-0.0, 0, 10]
gradient: [0, 0, 1]
```

Functions

Note: in Python, functions can ‘see’ variables outside of their own scope; when a variable is used in an expression, it first checks within its own scope, then it looks at the enclosing scope

```
In[2]: adv = 'This is an ad for ipython'
In[3]: def ad():
  ...:     print(adv)
  ...:
In[4]: ad()
This is an ad for ipython
```

Exercise

Create a function that optimizes over a convex quadratic function using gradient descent. Plot the function value during training using plot functions in Matplotlib library

The input quadratic function will be put in the form $f(\bar{x}) = \bar{x}^T \boldsymbol{A}\bar{x} + \bar{b}^T\bar{x} + c$

The function head should be

```
def quad_prog(A, b, c=0, enable_plot=True, max_itr=1000, step_size=0.01):
```

The function should return the optimal value and optimal solution

```
    return optim_val, optim_x
```

Hints:

1. You may assume that the input matrix **A** is symmetric
2. Gradient of a quadratic equation:$\nabla f_x = (A + A^T)\bar{x} + \bar{b}^T$
3. Gradient descent: $x^k = x^{k-1} - \eta \nabla f_x$

Basic data types and arithmetic operations

Numbers: only 3 intrinsic types – integer, double precision float, no single precision float, and complex

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x / 2)   # division; prints "1.5" in python 3.x and "1" in python 2.x; for integer division in python 3.x, use x //2 or int(x/2)
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
z = int(y) # convert from float to integer
print(z, type(z)) # Prints "2 <class 'int'>"
z = float(x) # convert from integer to float
print(z, type(z)) # Prints "8.0 <class 'float'>"
```

Materials on this page are adapted from CS231n Python Numpy Tutorial, originally prepared by Justin Johnson

Basic data types and arithmetic operations

Boolean type: Boolean in python is a subtype of integer, represented by reserved keywords: **True** and **False**

In addition to False object, all following are also evaluated to **false** in a truth value testing

- **None**
- zero of any numeric types: 0, 0.0, 0j
- any empty sequence: ‘’, (), []
- any empty mapping: {}
- instances of user-defined classes. If __len__() method returns zero or __bool__() method returns False

```
Python 3.6.5 (default, Apr 25 2018
In[2]: int(True)
Out[2]: 1
In[3]: int(False)
Out[3]: 0
In[4]: bool(0)
Out[4]: False
In[5]: bool(0.1)
Out[5]: True
```

```
In[2]: (True + True) ** (False)
Out[2]: 1
In[3]: ((True + False) * (3 * True)) ** (True + True)
Out[3]: 9
```

Basic data types and arithmetic operations

Booleans operations: Python uses English words (**and**, **or**, **not**) instead of symbols.

Both **and** and **or** operator follows short-circuit logic:

x and y: if x is false then x, else y

x or y: if x is true then x, else y

not operator has higher precedence than **and** and **or,** but lower precedence than other operators, so

```
In[6]: not 1 == 1
Out[6]: False
In[7]: 1 == not 1
  File "<ipython-input-7-3ad6336fcb7c>", line 1
    1 == not 1
           ^
SyntaxError: invalid syntax
```

Note: All Boolean results evaluated from an expression/function are either True/1 or False/0, except for **and** and **or.** Operator **and** and **or** always return one of its operands

Basic data types and arithmetic operations

Strings: there is no char type in python; a char is a string of length 1

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)  # prints "hello world 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; prints "  hello"
print(s.center(7))     # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another;
                                # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"
```

Materials on this page are copied from CS231n Python Numpy Tutorial, originally prepared by Justin Johnson

Basic data types and arithmetic operations

Containers: List, tuple, set, and dictionary

List: Elements contained in a list can be accessed with index. Python list index starts with 0. A list may contain elements of different data type. Lists are enclosed by square brackets []

Tuple: Creation and element access of tuple are the same as list. However, tuples are **immutable**; you cannot reassign a value to a tuple element. Also, tuples are hashable and can be used as keys for dictionary, lists cannot. Tuples are enclosed by parenthesis (). One can create a tuple with single element by appending a comma to the object ('helloTuple',)

```
nums = list(range(5))     # range is a built-in function that creates a list of integer
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])           # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive); prints "
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

Codes on this page are copied from CS231n Python Numpy Tutorial, originally prepared by Justin Johnson

Basic data types and arithmetic operations

Containers: List, tuple, set, and dictionary

Looping over list and tuple

```
In[13]: alist = list(range(2)) + ['car'] + [0.617] + [True]
In[14]: for a_element in alist:
   ...:     print(type(a_element), a_element)
   ...:
<class 'int'> 0
<class 'int'> 1
<class 'str'> car
<class 'float'> 0.617
<class 'bool'> True
```

```
In[15]: atuple = ('elephant', 1869, False, 2.33, (), [])
In[16]: for idx, a_element in enumerate(atuple):
   ...:     print(idx, type(a_element), a_element)
   ...:
0 <class 'str'> elephant
1 <class 'int'> 1869
2 <class 'bool'> False
3 <class 'float'> 2.33
4 <class 'tuple'> ()
5 <class 'list'> []
```

Basic data types and arithmetic operations

Containers: List, tuple, set, and dictionary

Set: A set represents a collection of **unordered**, **unique**, **immutable** objects. It cannot be indexed by subscripts, but can be iterated/looped over (in the same way as list and tuple).

```
In[32]: a_set = {'dog','cat'}
In[33]: a_set.add(3.14)
In[34]: print(a_set)
{3.14, 'cat', 'dog'}
In[35]: a_set.pop()
Out[35]: 3.14
In[36]: a_set = {'dog','cat'}
In[37]: a_set.add(3.14)
In[38]: print(a_set)
{3.14, 'cat', 'dog'}
In[39]: a_set.add('Siri')
In[40]: print(a_set)
{3.14, 'cat', 'dog', 'Siri'}
In[41]: a_set.pop()
Out[41]: 3.14
In[42]: a_set.pop()
Out[42]: 'cat'
```

```
In[43]: another_set = set(alist)
In[44]: print(another_set)
{0, 1, 0.617, 'car'}
In[45]: print(alist)
[0, 1, 'car', 0.617, True]
In[46]: for idx, ele in enumerate(another_set):
   ...:     print(idx, ele)
   ...:
0 0
1 1
2 0.617
3 car

In[47]:
```

Basic data types and arithmetic operations

Containers: List, tuple, set, and dictionary

Dictionary: A dictionary stores (key, value) pairs, where key can be any hashable object and value can be any arbitrary object.

```
In[2]: months = {'Jan':31, 'Feb':28, 'Mar':31}
In[3]: print(months['Feb'])
28
In[4]: months['April'] = 30
In[5]: print(months['April'])
30
In[6]: 'Mar' in months
Out[6]: True
In[7]: months.pop('Mar')
Out[7]: 31
In[8]: print(months)
{'Jan': 31, 'Feb': 28, 'April': 30}
```

```
In[9]: months['May']
Traceback (most recent call last):
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.fra
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-9-4e508fae9bf4>", line 1, in <module>
    months['May']
KeyError: 'May'
In[10]: months.get('May')
In[11]: months.get('May','N/A')
Out[11]: 'N/A'
In[12]: print(months.get('May'))
None
```

Basic data types and arithmetic operations

Containers: List, tuple, set, and dictionary

Looping over Dictionary

```
In[16]: d = {'1+1=':2, 'Purdue':['West Lafayette','IN','47907'], 0: True, 1: False, 3.14:'PI'}
In[17]: for key in d:
   ...:     print('key:',key,'value:',d[key])
   ...:
key: 1+1= value: 2
key: Purdue value: ['West Lafayette', 'IN', '47907']
key: 0 value: True
key: 1 value: False
key: 3.14 value: PI
In[18]: for key, val in d.items():
   ...:     print('key:',key,'| value:',val,'| type of val:',type(val))
   ...:
key: 1+1= | value: 2 | type of val: <class 'int'>
key: Purdue | value: ['West Lafayette', 'IN', '47907'] | type of val: <class 'list'>
key: 0 | value: True | type of val: <class 'bool'>
key: 1 | value: False | type of val: <class 'bool'>
key: 3.14 | value: PI | type of val: <class 'str'>
```

Classes

Python functions are defined using keyword **def**

```
class Greeter(object):

    # 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!"
```

Codes on this page are copied from CS231n Python Numpy Tutorial, originally prepared by Justin Johnson