Digital Signal Processing

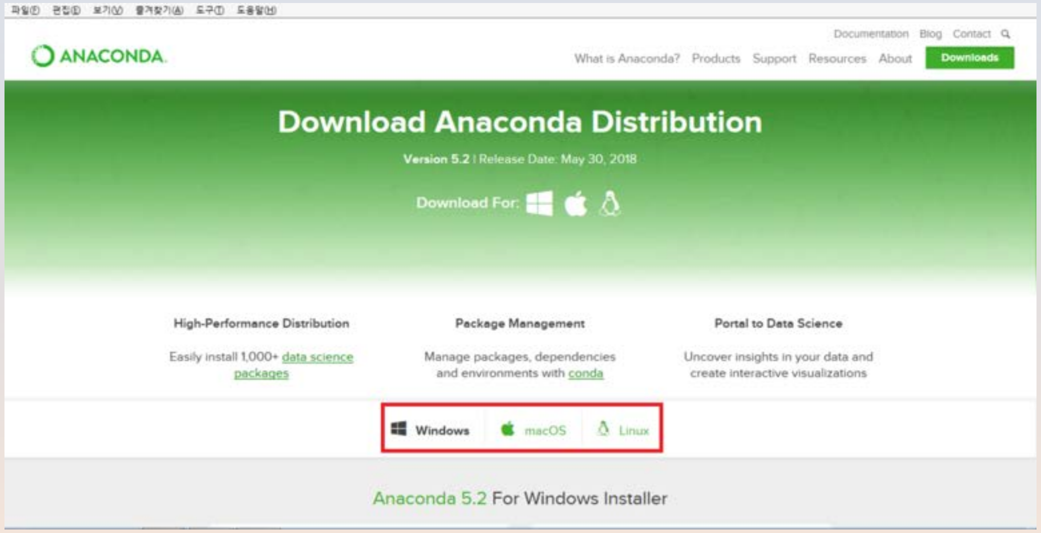
Lecture 2 – IDE & Python

상명대학교
컴퓨터과학과
강상욱 교수

Background

Software Install

- Use python 2.7 or python 3.6.
  - Python 3 introduced many backwards-incompatible changes.
  - http://python.org/ for documentation, tutorials, beginners guide, core distribution, …
- Use Anaconda, which is Python distribution that provides many packages.
  - https://www.anaconda.com
  - Use "Jupyter Notebook"


ANACONDA.
Documentation Blog Contact
What is Anaconda? Products Support Resources About Downloads
Download For:
Windows macOS Linux

Download Anaconda Distribution

Version 5.2 | Release Date: May 30, 2018

High-Performance Distribution

Easily install 1,000+ data science packages

Package Management

Manage packages, dependencies and environments with conda

Portal to Data Science

Uncover insights in your data and create interactive visualizations

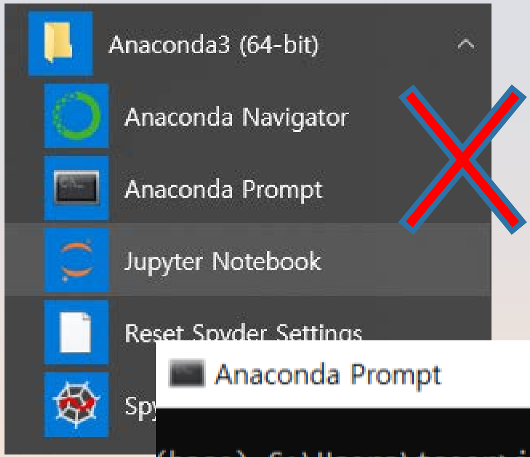
Anaconda 5.2 For Windows Installer

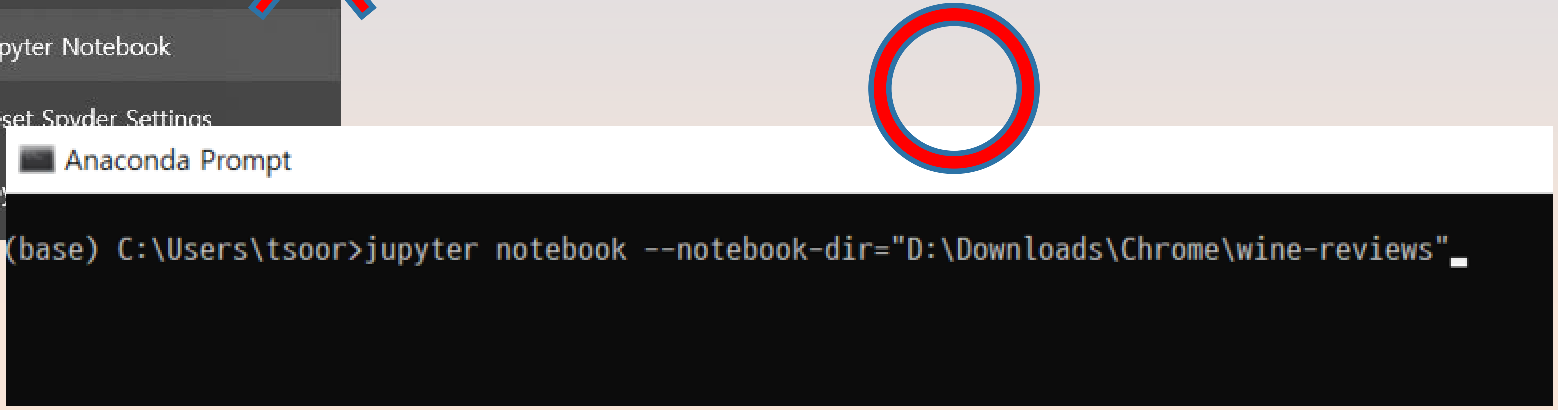
- The GitHub repository
  - https://github.com/AllenDowney/ThinkDSP.
  - You should fork it on GitHub and then clone it to your hard drive.
    - git clone [URL]

Jupyter Notebook I

- "notebook" or "notebook documents" denote documents that contain both code and rich text elements, such as figures, links, equations.
- As a server-client application, the Jupyter Notebook App allows you to edit and run your notebooks via a web browser (Chrome works best).

- 실행 – anaconda prompt 창을 통해서 실행



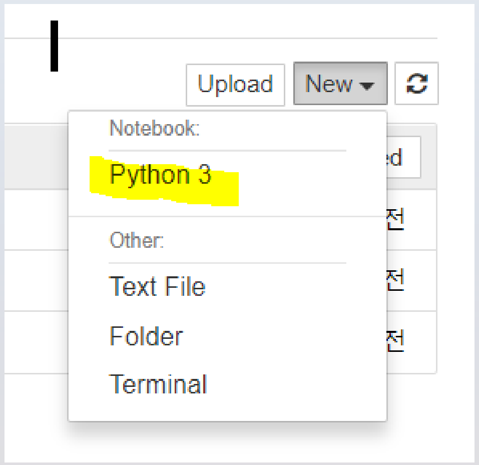


```
(base) C:\Users\tsoor>jupyter notebook --notebook-dir="D:\Downloads\Chrome\wine-reviews"
```

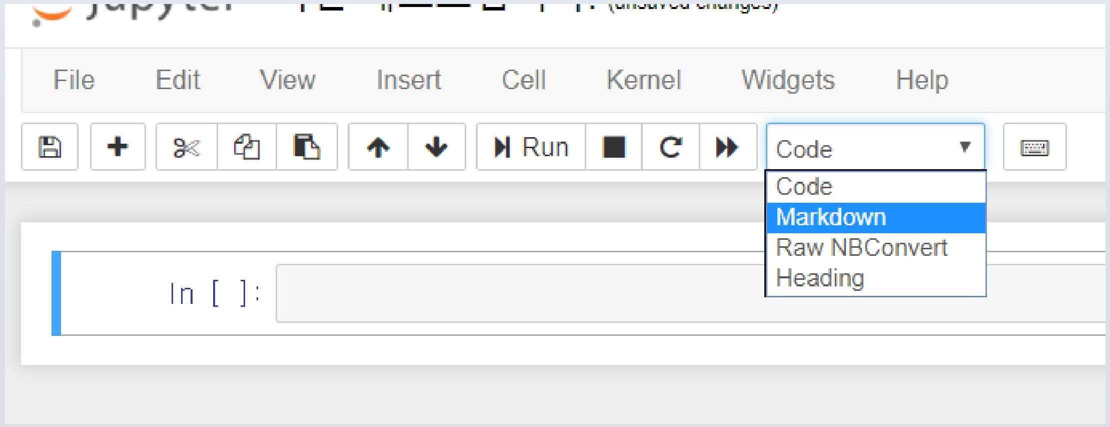
Jupyter Notebook 2

- Create a notebook

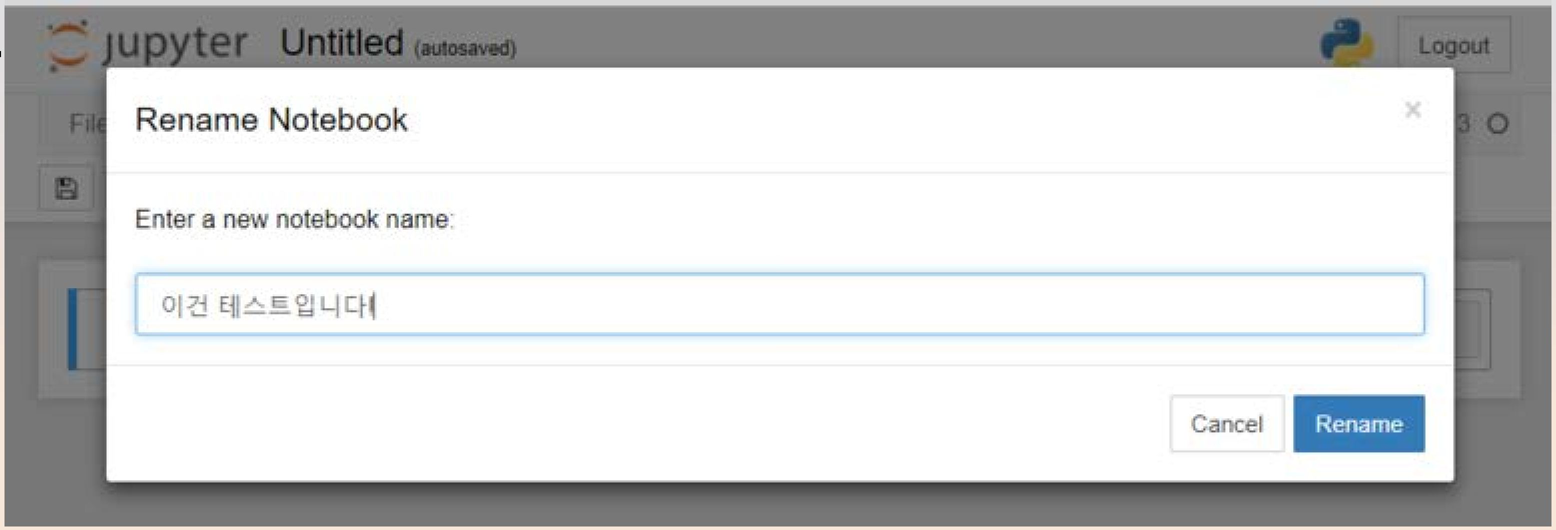
1



3



2



Enter a new notebook name: 이건 테스트입니다

4 Major Versions of Python

- "Python" or "CPython" is written in C/C++
  - Version 2.7 came out in mid-2010
  - Version 3.1.2 came out in early 2010
- "Jython" is written in Java for the JVM
- "IronPython" is written in C# for the .Net environment
- Check your Python version "python –version"

Development Environments

1. PyDev with Eclipse
2. Komodo
3. Emacs
4. Vim
5. TextMate
6. Gedit
7. Idle
8. PIDA (Linux)(VIM Based)
9. NotePad++ (Windows)
10. BlueFish (Linux)

what IDE to use? http://stackoverflow.com/questions/81584

Python Interactive Shell

```
% python
Python 2.6.1 (r261:67515, Feb 11 2010, 00:51:29)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can type things directly into a running Python session

```
>>> 2+3*4
14
>>> name = "Andrew"
>>> name
'Andrew'
>>> print "Hello", name
Hello Andrew
>>>
```

Data Types/Containers

Numbers

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

```
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)   # 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"
```

No unary increment & decrement
x++, x--

Booleans

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

```
t = True
f = False
print(type(t))  # Prints "<class 'bool'>"
print(t and f)  # Logical AND; prints "False"
print(t or f)   # Logical OR; prints "True"
print(not t)    # Logical NOT; prints "False"
print(t != f)   # Logical XOR; prints "True"
```

Strings 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"
```

Strings 2

```
smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles.find("(O)")
15
>>> smiles.find(".")
9
>>> smiles.find(".", 10)
-1
>>> smiles.split(".")
['C(=N)(N)N', 'C(=O)(O)O']

>>> names = [“Ben", “Chen", “Yaqin"]
>>> ", ".join(names)
‘Ben, Chen, Yaqin‘
```

Use “find” to find the start of a substring.

Start looking at position 10. Find returns -1 if it couldn’t find a match.

Split the string into parts with “.” as the delimiter

List

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

```
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
```

List - slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists.

```
nums = list(range(5))     # range is a built-in function that creates a list of integers
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 "[0, 1]"
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]"
```

List - loop

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

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.

animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

List - comprehension

When programming, frequently we want to transform one type of data into another.

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension, sometimes with conditions

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]

even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)  # Prints "[0, 4, 16]"
```

Dictionaries

A dictionary stores (key, value) pairs, similar to a Map in Java or an object in Javascript.

- Duplicate keys are not allowed
- Duplicate values are just fine

```
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"
d['fish'] = 'wet'        # Set an entry in a dictionary
print(d['fish'])         # Prints "wet"
print(d['monkey'])  # KeyError: 'monkey' not a key of d
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"
del d['fish']            # Remove an element from a dictionary
print(d.get('fish', 'N/A’))  # "fish" is no longer a key; prints "N/A"
```

Dictionaries – loop, item, comprehension

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

d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs“

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)  # Prints "{0: 0, 2: 4, 4: 16}"
```

Sets

A set is an unordered collection of distinct elements.

```
animals = {'cat', 'dog'}
print('cat' in animals)     # Check if an element is in a set; prints "True"
print('fish' in animals)    # prints "False"
animals.add('fish')         # Add an element to a set
print('fish' in animals)    # Prints "True"
print(len(animals))         # Number of elements in a set; prints "3"
animals.add('cat')          # Adding an element that is already in the set does nothing
print(len(animals))         # Prints "3"
animals.remove('cat')       # Remove an element from a set
print(len(animals))         # Prints "2"
```

Sets – loop, comprehension

A set is an unordered collection of distinct elements.

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

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

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.

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)          # Create a tuple
print(type(t))      # Prints "<class 'tuple'>"
print(d[t])         # Prints "5"
print(d[(1, 2)])    # Prints "1"
```

Lists are mutable - some useful methods

```
>>> ids = ["9pti", "2plv", "1crn"]
>>> ids.append("1alm")
>>> ids
['9pti', '2plv', '1crn', '1alm']
>>>ids.extend(L)
```

Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

```
>>> del ids[0]
>>> ids
['2plv', '1crn', '1alm']
>>> ids.sort()
>>> ids
['1alm', '1crn', '2plv']
>>> ids.reverse()
>>> ids
['2plv', '1crn', '1alm']
>>> ids.insert(0, "9pti")
>>> ids
['9pti', '2plv', '1crn', '1alm']
```

append an element

remove an element

sort by default order

reverse the elements in a list

insert an element at some specified position.
(Slower than .append())

Functions/ Classes

Functions

Python functions are defined using the ‘def’ keyword.

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

Functions with optional keyword arguments

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

Classes

```
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!"
```

Numpy

Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the ‘rank’ of the array; the ‘shape’ of an array is a tuple of integers giving the size of the array along each dimension..

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Arrays – functions to create arrays

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7) # Create a constant array
print(c)              # Prints "[[ 7.  7.]
                      #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"
e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

Arrays Indexing – slicing 1

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.

```
import numpy as np
# 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]

# A slice of an array is a view into the same data, so modifying it will modify original array.
print(a[0, 1])     # Prints "2"
b[0, 0] = 77       # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])     # Prints "77“  , cf> np.copy( )
```

Arrays Indexing – slicing 2

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the original array:
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
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)   # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)   # Prints "[[ 2]
                              #          [ 6]
                              #          [10]] (3, 1)"
```

Integer Arrays Indexing

When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array.

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing. The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])     # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))        # Prints "[1 4 5]"

# When using integer array indexing, you reuse the same element from the source array:
print(a[[0, 0], [1, 1]])        # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))        # Prints "[2 2]"
```

```
import numpy as np
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])       # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)    # prints "array([[11,  2,  3],
            #                [ 4,  5, 16],
            #                [17,  8,  9],
            #                [10, 21, 12]])
```

Boolean Arrays Indexing

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
import numpy as np

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

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array consisting of the elements
# of a corresponding to the True values of bool_idx
print(a[bool_idx])     # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])        # Prints "[3 4 5 6]"
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

Array math

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

Note that * is elementwise multiplication, not matrix multiplication. We instead 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:

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

# Elementwise sum; both produce the array    [[ 6.0  8.0]
#                                             [10.0 12.0]]
print(x + y)
print(np.add(x, y))
# Elementwise difference; both produce the array    [[-4.0 -4.0]
#                                                    [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
# Elementwise product; both produce the array    [[ 5.0 12.0]
#                                                 [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
# Elementwise division; both produce the array    [[ 0.2         0.33333333]
#                                                  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
# Elementwise square root; produces the array    [[ 1.          1.41421356]
#                                                 [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
import numpy as np
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))

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

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

x = np.array([[1,2],[3,4]])
print(np.sum(x))            # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Transpose

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
             #          [3 4]]"
print(x.T)  # Prints "[[1 3]
             #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)     # Prints "[1 2 3]"
print(v.T)   # Prints "[1 2 3]"
```

Broadcasting – Normal way

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

```
import numpy as np
# We will add the vector v to each row of the matrix x, storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)    # Create an empty matrix with the same shape as x
# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v
# Now y is the following    [[ 2  2  4]
#                            [ 5  5  7]
#                            [ 8  8 10]
#                            [11 11 13]]
print(y)
```

Broadcasting – Stacking

however when the matrix x is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv.

```
import numpy as np
# We will add the vector v to each row of the matrix x, storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv    # Add x and vv elementwise
print(y)   # Prints "[[ 2  2  4
           #          [ 5  5  7]
           #          [ 8  8 10]
           #          [11 11 13]]"
```

Broadcasting – Broadcasting

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v.

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v     # Add v to each row of x using broadcasting
print(y)      # Prints "[[ 2  2  4]
              #          [ 5  5  7]
              #          [ 8  8 10]
              #          [11 11 13]]"
```

SciPy

SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

Image operations

SciPy provides some basic functions to work with images. For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images.

```
from scipy.misc import imread, imsave, imresize
# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)        # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels by a different scalar
# constant. The image has shape (400, 248, 3);  we multiply it by the array [1, 0.95, 0.9] of
# shape (3,); numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9 respectively.
img_tinted = img * [1, 0.95, 0.9]
# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))
# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

Distance between points

SciPy defines some useful functions for computing distances between sets of points.

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

# Create the following array where each row is a point in 2D space:   [[0 1]
#                                                                       [1 0]
#                                                                       [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :], and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

Matplotlib

Plotting - Basic

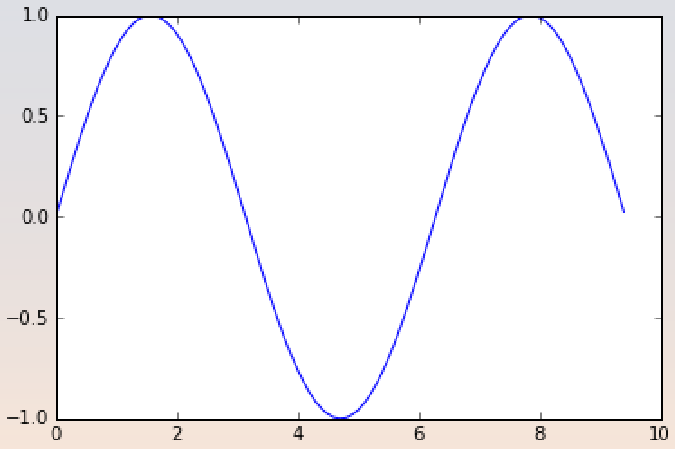
Matplotlib is a plotting library. In this section give a brief introduction to the matplotlib.pyplot module, which provides a plotting system similar to that of MATLAB.

The most important function in matplotlib is plot, which allows you to plot 2D data.

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

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()       # You must call plt.show() to make graphics appear.
```



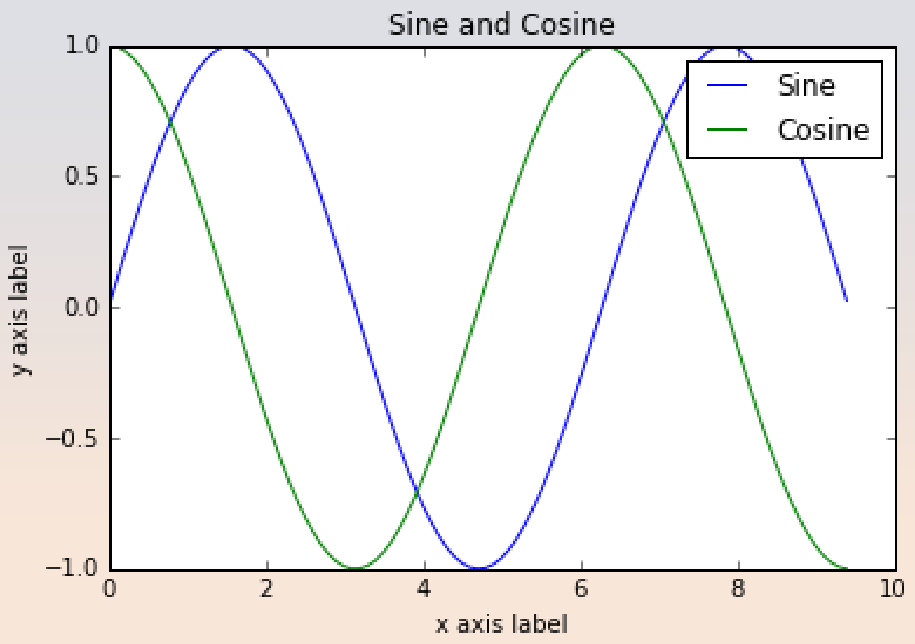
Plotting - Rich

With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels.

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

# 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)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

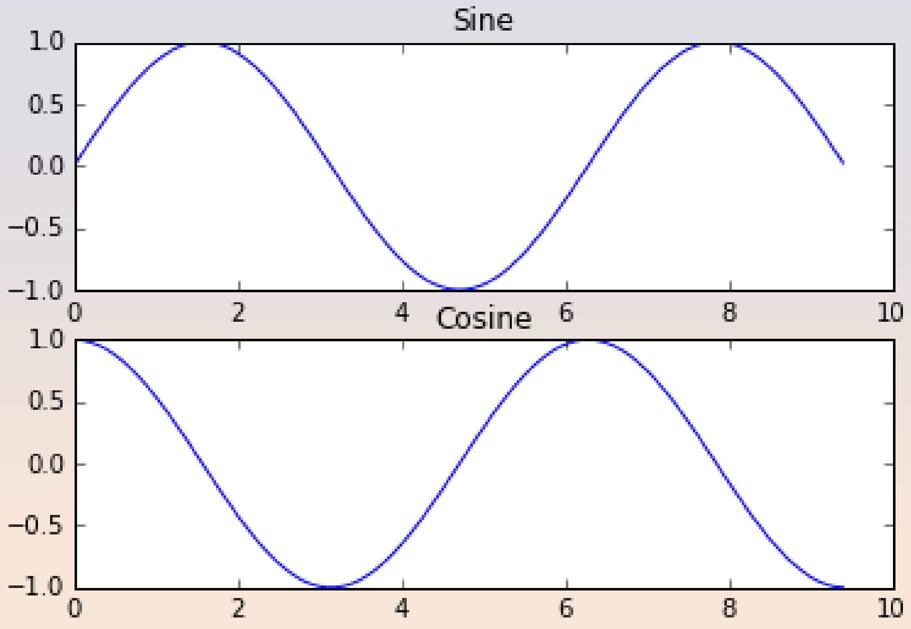


Plotting - Subplots

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

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')
# Show the figure.
plt.show()
```



Images

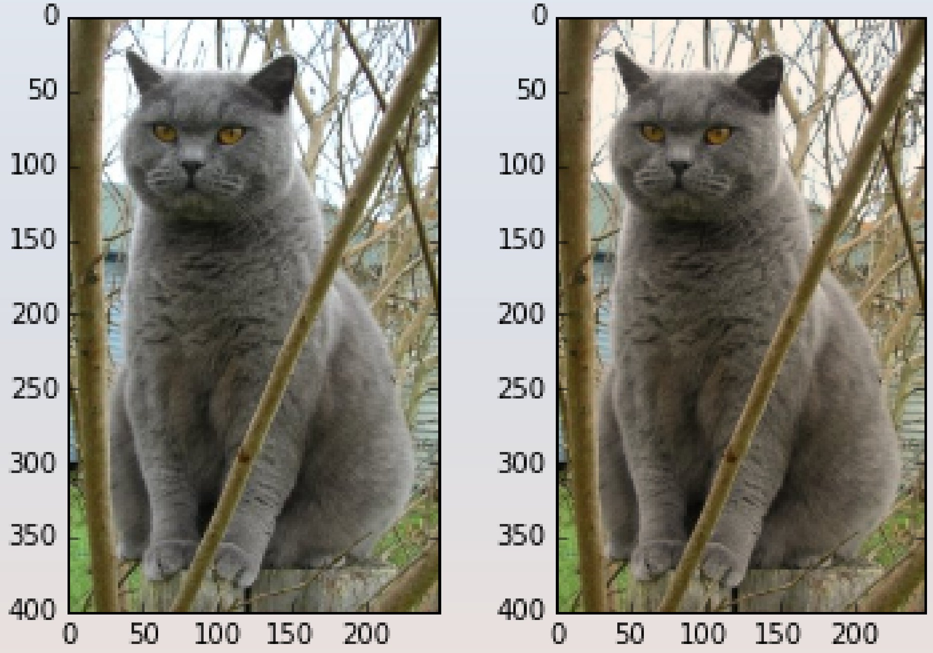
```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results if presented with
# data that is not uint8. To work around this, we explicitly cast the image to uint8
# before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```



Miscellaneous

zipping lists together

```
>>> names
['ben', 'chen', 'yaqin']

>>> gender = [0, 0, 1]

>>> zip(names, gender)
[('ben', 0), ('chen', 0), ('yaqin', 1)]
```

Control Flow

Things that are False

- The boolean value False
- The numbers 0 (integer), 0.0 (float) and 0j (complex).
- The empty string "".
- The empty list [], empty dictionary {} and empty set set().

Things that are True

- The boolean value True
- All non-zero numbers.
- Any string containing at least one character.
- A non-empty data structure.

Reading files

```
>>> f = open(“names.txt")
>>> f.readline()
'Yaqin\n’
>>> lst= [ x for x in open("text.txt","r").readlines() ]
>>> lst
['Chen Lin\n', 'clin@brandeis.edu\n', 'Volen 110\n', 'Office Hour: Thurs. 3-5\n', '\n', 'Yaqin
    Yang\n', 'yaqin@brandeis.edu\n', 'Volen 110\n', 'Offiche Hour: Tues. 3-5\n']
```

Ignore the header?

```
for (i,line) in enumerate(open(‘text.txt’,"r").readlines()):
    if i == 0: continue
    print line
```

File Output

```
input_file = open(“in.txt")
output_file = open(“out.txt", "w")
for line in input_file:
        output_file.write(line)
```

“w” = “write mode”
“a” = “append mode”
“wb” = “write in binary”
“r” = “read mode” (default)
“rb” = “read in binary”
“U” = “read files with Unix or Windows line endings”

import __future__

- A directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python.
- The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language.
- It allows use of the new features on a per-module basis before the release in which the feature becomes standard.
- Example
  - In Python 2.7 : 5/2=2
  - In Python 3.x : 5/2=2.5
  - So, use
  - **from** __future__ **import** division
  - c = 5/2   *# c = 2.5*