NYC DATA SCIENCE
ACADEMY

Week 1: Get Started with Python

NYC Data Science Academy

OVERVIEW

- Installing and using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter

Introduction to Python

- Python has become one of the most widely used programming languages, especially in the domain of web computing.
  - Python makes it simple and easy to write small programs, called *scripts*.
  - It is especially good at handling text, which makes it ideal for “scraping” web pages.
  - It has a huge collection of libraries, including ones that provide for graphics, numerical processing, data transformations, and machine learning.

Python makes simple things simple

C++

```cpp
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello World" << endl;
   return 0;
}
```

Java

```java
public class HelloWorld{

     public static void main(String []args){
        System.out.println("Hello World");
     }
}
```

Python

```python
print 'Hello World'
```

This class

- This is a four-week introduction to Python. We will concentrate on the basics of the language. We will also introduce some widely-used packages, such as numpy, for numerical processing, and pandas for data manipulation.
- We will work in a highly interactive way, with numerous in-class exercises.
- You will do your coding in Python via *IPython notebooks*. This provides a convenient interface for both editing and executing Python code.
  - If you use Python at work, they may require you to use a different operating environment, such as the Unix command line. But that will be easy to learn compared to learning the Python language.

OVERVIEW

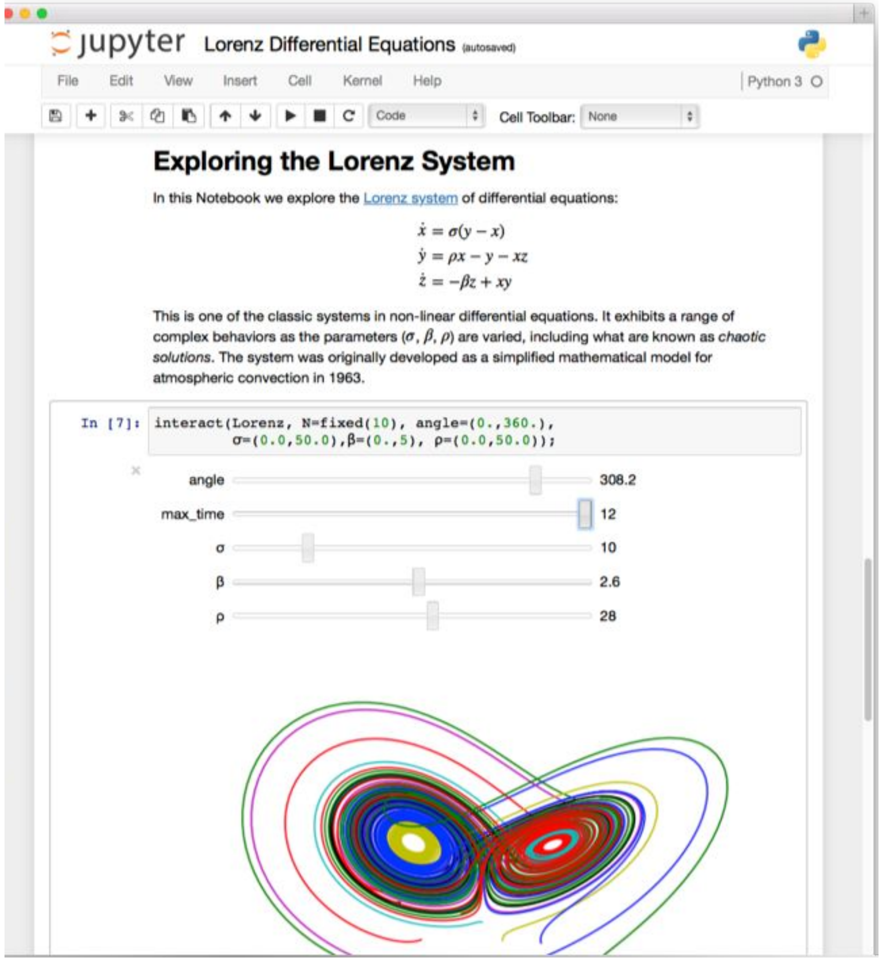
- Installing and using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter

How to run Python code

- There are multiple ways to run Python, but two are most common:
  - Write all the code using a text editor like vi or Sublime Text and run the .py script using the command line. This is the method used by most programmers.
  - IPython Notebook, a system in which you can edit and execute Python scripts.

IPython Notebook

- IPython Notebook: an interactive computational environment
  - you can combine code execution, rich text, mathematics, plots and rich media.


jupyter
Lorenz Differential Equations (autosaved)
File Edit View Insert Cell Kernel Help
Python 3
Code
Cell Toolbar: None

Exploring the Lorenz System

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

This is one of the classic systems in non-linear differential equations. It exhibits a range of complex behaviors as the parameters (σ, β, ρ) are varied, including what are known as *chaotic solutions*. The system was originally developed as a simplified mathematical model for atmospheric convection in 1963.

```
In [7]: interact(Lorenz, N=fixed(10), angle=(0.,360.),
                 σ=(0.0,50.0),β=(0.,5), ρ=(0.0,50.0));
```

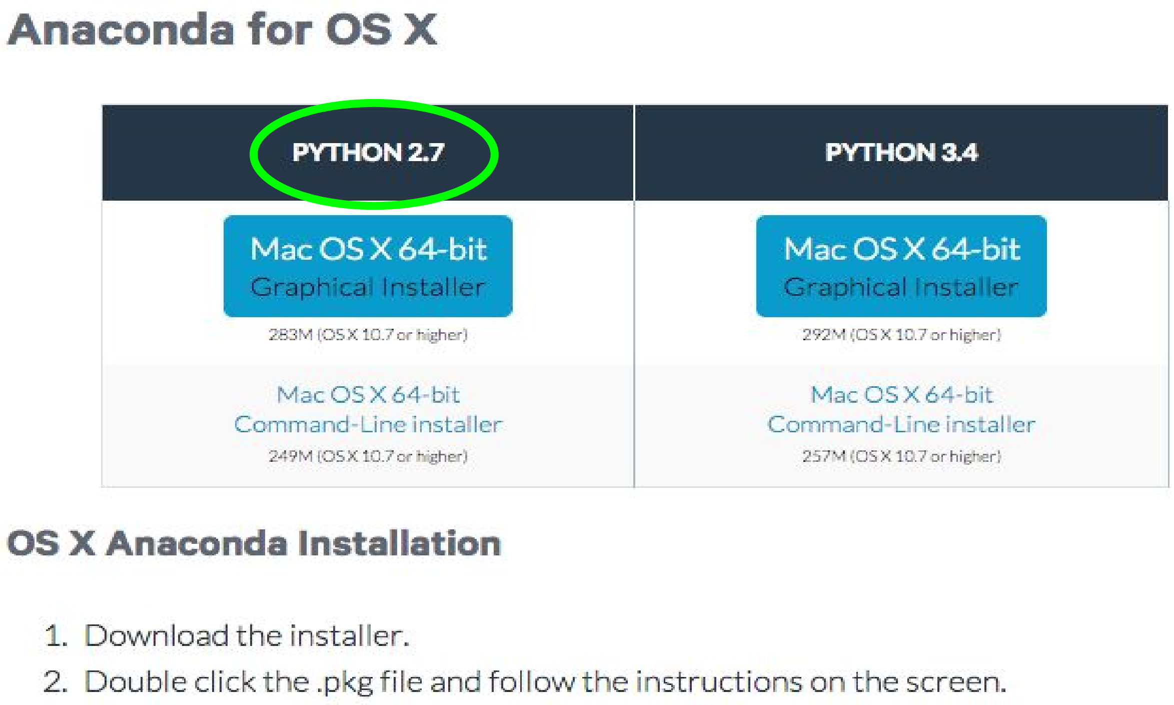
```
angle     308.2
max_time  12
σ         10
β         2.6
ρ         28
```

How to install IPython Notebook

- The general way is to install it through pip, which is the Python package manager. Since we haven't covered it yet, let's just "cheat" a little bit here. For this class, we will use Anaconda.
- We usually call Anaconda the "one-stop shop" for data science because it has all the useful packages pre-installed, including numpy, scipy, pandas and definitely IPython Notebook. It is very helpful especially when you are new to Python.
- **If you don't have anaconda installed, you can use a free version of IPython by going to:** try.jupyter.org

How to install Anaconda

- Let’s download Anaconda from here. First choose the version of your operating system and then select Python 2.7. The installation process should be straightforward. For example:



Anaconda for OS X

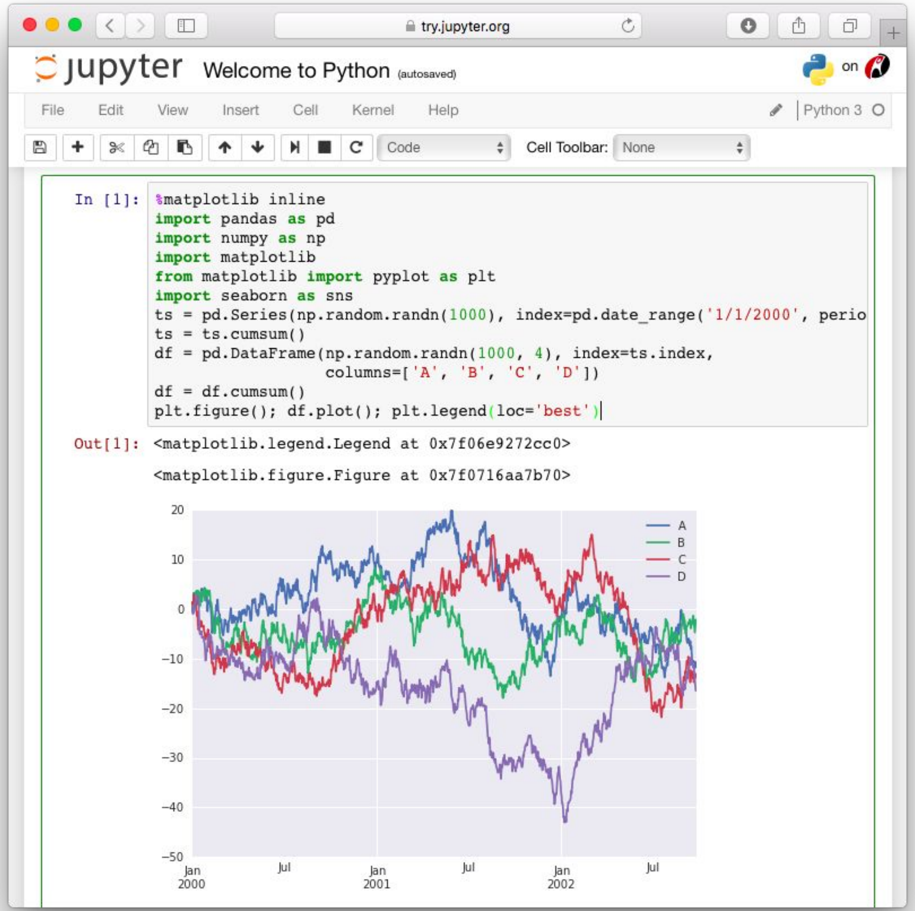
PYTHON 2.7	PYTHON 3.4
Mac OS X 64-bit Graphical Installer 283M (OS X 10.7 or higher)	Mac OS X 64-bit Graphical Installer 292M (OS X 10.7 or higher)
Mac OS X 64-bit Command-Line installer 249M (OS X 10.7 or higher)	Mac OS X 64-bit Command-Line installer 257M (OS X 10.7 or higher)

OS X Anaconda Installation

1. Download the installer.
2. Double click the .pkg file and follow the instructions on the screen.

How to use IPython notebook

- Your code goes into the gray box, or "cell".
- Click ➤ or press "Shift+Enter" to run the highlighted cell.
- When a cell is running, it will shown as `In [*].`
- When it's done, the output will be shown below, and an empty cell will appear after it.
- To stop a running cell, click ■ "interrupt kernel"
- *Tips*: You can move / modify / delete an existing cell


try.jupyter.org
jupyter Welcome to Python (autosaved)
on
File Edit View Insert Cell Kernel Help
Python 3
Code
Cell Toolbar: None
A
B
C
D

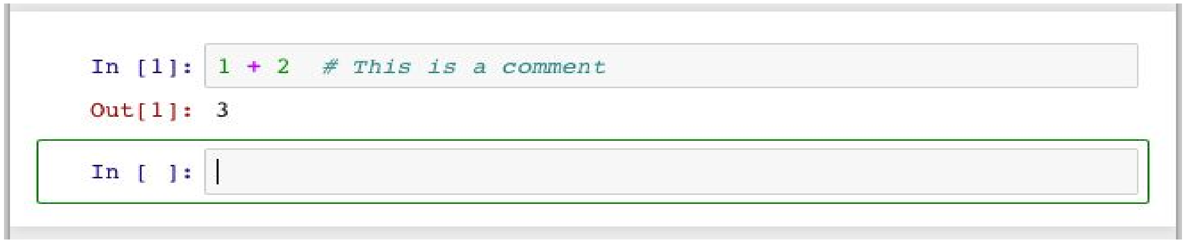
```
In [1]: %matplotlib inline
        import pandas as pd
        import numpy as np
        import matplotlib
        from matplotlib import pyplot as plt
        import seaborn as sns
        ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', perio
        ts = ts.cumsum()
        df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                          columns=['A', 'B', 'C', 'D'])
        df = df.cumsum()
        plt.figure(); df.plot(); plt.legend(loc='best')

Out[1]: <matplotlib.legend.Legend at 0x7f06e9272cc0>

        <matplotlib.figure.Figure at 0x7f0716aa7b70>
```

OVERVIEW

- Installing and using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter

Simple values and expressions

- Python is used interactively. Normally, you enter Python expressions and it responds with a value. Thus, it can be used as a calculator.



```
In [1]: 1 + 2  # This is a comment
Out[1]: 3
In [ ]:
```

- The figure above shows what you can see in iPython Notebook. Hereafter we will put the demo code in a gray textbox so you can copy and paste in your ipython notebook and run it. Now try:

```
1 + 2  # This is a comment
3
```

- *Note*: Comments in Python start with the hash character, #, and extend to the end of the physical line and we mark the output in red.

Simple values and expressions

- Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages; parentheses `( )` can be used for grouping.

```
1 + 2 * 3     # 1+2*3 = 7
(1 + 2) * 3   # (1+2)*3 = 9
```

- *Note*: use `print` statement to output multiple results from one single cell
- Now try more examples and check the output:

```
17 / 3   # int / int -> int
17 / 3.0   # int / float -> float
17 // 3.0  # explicit floor division print
17 % 3   # remainder
2 ** 7   # 2 to the power of 7
```

Simple values and expressions

❖ Syntactic note: Python does not have any special terminating character , so you must enter everything on one line. If the line is too long, you can split it up either by enclosing the expression in parentheses or by adding a backslash at the end:

```
print (8 * (7 + 6 * 5) + 4 / 3 ** 2 - 1)
295
print (8 * (7 + 6 * 5)     # use parentheses
       + 4 / 3 ** 2 - 1)
295
print 8 * (7 + 6 * 5) \
      + 4 / 3 ** 2 - 1       # use backslash
295
```

❖ *Note*: \ should be the end of a line, and therefore you cannot have comment or even spaces on that line

Variables

- Variables are used to give names to values. This is to avoid having to re-type an expression, and so the computer doesn't have to recompute it. The equal sign (=) is used to assign a value to a variable

```
tax = 12.5 / 100
price = 100.50
price * tax
12.5625
```

- The **last printed expression** is assigned to _.

```
price + _        # _ = 12.5625
113.0625
```

Calling functions

❖ The Python interpreter has a number of functions built into it that are always available. One example is the absolute value function:

```
abs(-5.0)
5.0
```

❖ Python has a way to put definitions in a file that you can load and use. Such a file is called a *module*. You use the functions in a module by importing the module and using its name plus the function's name:

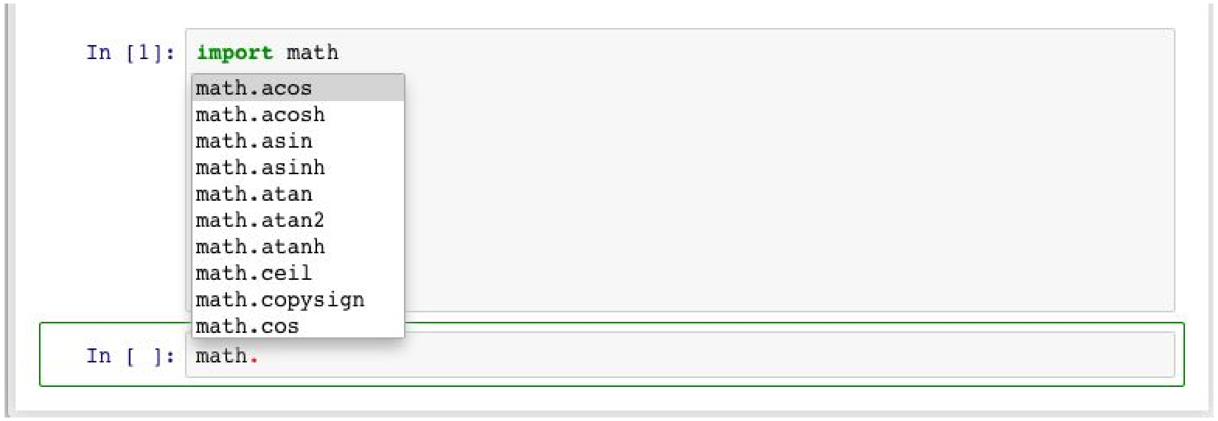
```
import math             # import the math module
math.factorial(5)       # factorial of 5
```

Or, use a different import syntax and use the function name alone:

```
from math import *
factorial(5)
```

How can I find the functions?

- Documentation
  - built-in functions:
    - https://docs.python.org/2/library/functions.html#pow
  - math module:
    - https://docs.python.org/2/library/math.html
- Google is your friend!
- And also the **Tab** key



```
In [1]: import math

In [ ]: math.
```

```
math.acos
math.acosh
math.asin
math.asinh
math.atan
math.atan2
math.atanh
math.ceil
math.copysign
math.cos
```

Naming conventions

- Variable names in Python should start with letters, and can contain any number of letters, digits, and _.
- Python names are case sensitive. This applies both to your variable name and to the function names imported from modules.
- By convention, Python variables usually start with lower-case letters. Variables should have descriptive names; for multi-word names, separate the words by underscores.
  - Good names: `i`, `j`, `first_index`, `random_nums`

Exercise 1: Exploring IPython

❖ In the input panel, run Python commands:

1. Calculate 17 / 3
2. Calculate 17 / -3 and compare it with the previous result.
3. Calculate 5 to the power of 3.
4. Import the math module and find a function to calculate the square root (the function name starts with “s”) of the last expression using _. Assign it to a variable (give it a name).
5. Calculate the square of that variable; does it differ from the result of the previous question?

Multiple expressions in iPython cells

- Our practice thus far has been to enter a single expression and then shift-enter. This evaluates the expression and opens a new cell.
- You can put multiple expressions in a single cell. All are evaluated, but *only the last has its value printed*.
- If you want to see the values of multiple expressions in a cell, add the word "print" before the expressions (all but the last). You hit shift-enter just once and all the expressions in the cell are evaluated.
- This does not apply to variable assignment. When you assign an expression to a variable, its value is not normally printed anyway; if you want to know its value, enter the variable alone as an expression.

OVERVIEW

- Installing and using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter

Defining functions

- You know how to call functions. Now we'll learn how to define functions.
- This function takes a number x and returns $x^2 + x^3$.

```
def add_two_powers(x):
    return x**2 + x**3
```

- The keyword **def** introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and **must be indented**.
- To call the function, you do (shouldn't be indented if in the same cell)

```
add_two_powers(4)
80
```

Lambda expressions

- There is an alternative syntax for function definitions that will turn out to be handy. It uses the **lambda** keyword.
  - Back to the previous problem, we could write:

```
add_two_powers = lambda x: x**2 + x**3
add_two_powers(4)
80
```

- For most of today's class, I will ask you to define functions in both forms, just for the practice.

def vs. lambda

- The two syntaxes for function definitions give the same result, but note the differences in syntax:
  - The first version always begins with “def” (this is called a “keyword” because it is a special word in the syntax; other special words we’ve seen are “return” and “lambda”), and then a name and then the arguments in parentheses. The second version begins with the function name, an equal sign, the keyword “lambda”, and the arguments *not* in parentheses
  - The first version includes the keyword “return”; the second version doesn’t.

Defining functions with more than one argument

❖ To define a function with multiple arguments, just add more names to the variable list, separated by commas. This works the same in both notations.

```
import math

def vector_length_1(x, y):
    return math.sqrt(x**2 + y**2)

vector_length_2 = lambda x, y: math.sqrt(x**2 + y**2)
```

Exercise 2: Defining functions

- Define a function that takes the radius of a circle as input and return the area. (Remember the area of a circle = πr^2.) Define it with both syntaxes, `def` and `lambda`, calling them `area` and `Area`, respectively:

```
def area(x):
    ...

Area = lambda x: ...
```

  You need to use `math.pi`, which is defined in the math module.

- Calculate the area of a circle with radius 10 using both functions.

OVERVIEW

- Installing and using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter

Lists

- The power of Python comes from having values other than numbers. The two other types of values we'll use most often are lists and strings.
  - Lists are ordered collections of values.
    - Examples: [1, 2, 3, 4], [7.5, 9.0, 2]
  - Strings are sequences of characters
    - Examples: 'This class is about Python.' (Note that the space is considered a character.)
- Strings are actually not much more than lists of characters. We will look at lists first.
- *Syntactic note*: Strings can be written with either single or double quotes.

Lists of strings

❖ Lists can contain any types of values, including strings.

❖ Having lists of string is very useful, for this reason: You can read an entire file into a list, with each line being treated as a string. The length of the list will be exactly the same as the number of lines in the file. Then you can do any operations you want on the file using the list-manipulating power of Python.

  ➢ We will do exactly that next week, after we've learned more about strings. This week, we're concentrating on learning about lists. By the time we read files, we'll be able to do a lot with them.

List operations and functions

- You can manipulate lists in Python, meaning you can create new lists or get values out of lists. We'll spend some time going over the major list operations provided by Python.
- As we've seen, you create a list by writing values in square brackets, separated by commas.
  - Note that you can also have a list with no elements, written []. This list is astonishingly useful - kind of like the number zero.
- Given two lists, use + to concatenate them:

```
squares = [1,4,9,16,25]
alphabet = ['a','b','c','d']
squares + alphabet
[1, 4, 9, 16, 25, 'a', 'b', 'c', 'd']
```

List operations and functions

- Find the length of a list using the len() function:

```
len(squares)
5
len(alphabet)
4
len(squares + alphabet)
9
```

- The function range gives a list of integers:

```
range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(10, 15)
[10, 11, 12, 13, 14]
range(15, 10, -1)
[15, 14, 13, 12, 11]
```

List operations and functions

❖ To create a list just one longer than an existing list - i.e. add an element to the end of a list - use concatenation:

```
squares + [36]
[1, 4, 9, 16, 25, 36]
```

But note: This operation does not change squares. That is, it does not "add 36 to squares," but rather creates a new list that has the same elements as squares, plus one more.

List operations and functions

❖ There are several operations that apply to lists of numbers and perform operations on the entire list:

➢ sum: Take the sum of the numbers in a list

➢ max, min: Take the maximum or minimum of the numbers in a list. (This can also apply to strings, using lexicographic order.)

```
sum(squares)
55
max(alphabet)
'd'
```

➢ sorted: Sorts a list:

```
sorted([4, 3, 6, 2, 5])
[2, 3, 4, 5, 6]
```

Subscripting

❖ Get a single element of a list by subscripting with a number. We number the elements from zero (which is pretty common for computer languages, but takes some getting used to). We can also subscript from the end by using negative numbers (-1 being the last element).

```
squares[3]
16
squares[-1]
25
alphabet[6]   #Be careful about this one
IndexError: list index out of range
```

Subscripting

- Subscripting can also be used to get more than one element of a list (called a "slice" of a list). E.g. list[*m*:*n*] returns a list that has the *m*th through (*n*-1)th element of list (again, counting from zero).

```
squares[1:3]
[4, 9]
alphabet[2:]
['c', 'd']
```

Exercise 3: List operations

- Define x to be the list [1, 2, 3, 4, 3, 2, 1]. Do this by using range twice and concatenating the results.
- Use subscripting to select both 2's from x.
- Define y to be [1, 2, 3, 4, 5, 4, 3, 2, 1]. Do this by separating out the first four elements of x (using subscripting) and the last three elements of x, and concatenating these together with [5, 4] in between.

Defining functions on lists

- Functions are defined on lists in exactly the same way as on numbers. (We continue defining every function in both syntaxes, just for practice.)
- A function that returns the first element of a list:

```
def firstelt(L):
    return L[0]

Firstelt = lambda L: L[0]

firstelt(squares)
1
Firstelt(squares)
1
```

Defining functions on lists

❖ A function that takes a list L and a value v, and returns a list that is the same as L except that its first element is v.

```
def replacefirst(L, v):
    return [v] + L[1:]

Replacefirst = lambda L, v:  [v] + L[1:]

replacefirst(squares, 7)
[7, 4, 9, 16, 25]
```

Defining functions on lists

❖ A function that has an integer argument n, and returns a list containing n, n+1, and n+2:

```
def threevals(n):
    return [n, n+1, n+2]

Threevals = lambda n: [n, n+1, n+2]
```

❖ A function that takes a list L and returns a list containing the first, third, and fifth elements of L.

```
def list135(L):
    return [L[0], L[2], L[4]]

List135 = lambda L: [L[0], L[2], L[4]]
```

Exercise 4: List operations

❖ Then define the following functions. Define them with both notations, changing the first letter of the name to a capital to distinguish them:

1. A function sum2 that returns the sum of the first two elements of a list:

```
sum2([4, 7, 9, 12])
11
```

2. A function that concatenates a list to itself:

```
double_lis([4, 7, 9, 12])
[4, 7, 9, 12, 4, 7, 9, 12]
```

OVERVIEW

- ❖ Installing and using iPython
- ❖ Simple values and expressions
- ❖ Lambda functions and named functions
- ❖ Lists
    - ➢ Built-in functions and subscripting
    - ➢ **Nested lists**
- ❖ Functional operators: map and filter

Nested lists

- Lists can contain any type of values, *including other lists*. This can be confusing, but the principle is the same as with any other elements.
- The list [v1, v2, …, vn] has n elements. The first is v1, the second is v2, etc. This is true no matter what the type of the elements is:
    - [1, 2, 3] has three elements. (Try len([1,2,3]).)
    - [1, 2, [3, 4, 5]] also has three elements.
    - [1, ["ab", "cd"], [3, 4, 5]] also has three elements.
    - [[], ["ab", "cd"], [3, 4, 5]] also has three elements.
- On the next page are more examples. Understanding nested lists is very important, and we'll take as much time as we need for you to get it.

Nested lists

❖ The operations we’ve seen above like “+”, len( ) and subscripting work the same on nested lists as on non-nested lists.

```
x = [[], 1, ['ab', 'cd'], [3, 4, 5]]
len(x)
4
y = [['q', 'r', 's'], 2]
len(y)
2
x[2]
['ab', 'cd']
(x[2])[0]
'ab'
x + y
[[], 1, ['ab', 'cd'], [3, 4, 5], ['q', 'r', 's'], 2]
```

Exercise 5: Nested lists

❖ Write a function `firstfirst` (in both syntaxes) that takes a nested list as input and returns the first element of the first element. (The first element must be a non-empty list).

```
y = [['q', 'r', 's'], 2]
firstfirst(y)
'q'
```

❖ Write a function `subscr2` that takes two arguments: a nested list and a list with two integers. It uses those two integers as indexes into the nested list:

```
y = [['q', 'r', 's'], 2]
subscr2(y, [0, 2])   # return y[0][2]
's'
```

OVERVIEW

- Installing and using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
    - Built-in functions and subscripting
    - Nested lists
- Functional operators: map and filter

map: Applying functions to all elements of a list

- You will often want to apply a function to all elements of a list.
  - Next week, we will read in files and turn them into lists. Applying a function to each element of the list will be applying a function to each line of the file, which is a commonly done thing.
- Suppose a list L has elements that are all numbers, and f is a function on numbers. Then map(f, L) is the result of applying f to every element of L.

```
L = [4, 9, 16]
map(math.sqrt, L)
[2.0, 3.0, 4.0]
map(add_two_powers, L)
[80, 810, 4352]
map(Add_two_powers, L)
[80, 810, 4352]
```

map: Applying functions to all elements of a list

- You can apply any one-argument function, as long as the values in the list are of the type that function expects.

```
nested_list1 = [[], ["ab", "cd"], [3, 4, 5]]
map(len, nested_list1) #len applies to lists
[0, 2, 3]


map(threevals, [1, 10, 20]) # three_vals applies to numbers
[[1, 2, 3], [10, 11, 12], [20, 21, 22]]

# firstelt applies to lists that are of non-zero length
map(firstelt, nested_list1)
IndexError: list index out of range


map(firstelt, nested_list1[1:])
['ab', 3]
```

Defining functions that use map

❖ Functions can use map just as they can use any other function.

```
def get_lengths(L):
    return map(len, L)

Get_lengths = lambda L: map(len, L)
Get_lengths(nested_list1)
[0, 2, 3]

def get_first_elements(L):
    return map(firstelt, L)

Get_first_elements = lambda L: map(firstelt, L)
get_first_elements(nested_list1[1:])
['ab', 3]
```

Exercise 6: map

❖ Define these functions.

1. square_all squares every element of a numeric list. (Define sqr to square a number, and map it over the list.)

```
square_all([1, 2, 3])
[1, 4, 9]
```

2. get_second gets the second element of every list in a nested list. (Define snd to get the second element of a list, and map it.)

```
get_second([[1, 2, 3], [4, 5], [6, 7, 8]])
[2, 5, 7]
```

3. tot_length finds the total length of the lists in a nested list.

```
tot_length([[1, 2, 3], [4, 5], [6, 7, 8]])
8
```

Boolean functions

- Boolean functions return one of the truth values True or False.
- There are built-in operators and functions that return booleans:
  - Arithmetic comparison operators: ==, !=, …
- Conditions can be combined using and, or, and not:

```
x = 6

x > 5 and x < 7
True

x != -1 or x >= 10
True
```

Defining boolean functions

- Defining boolean functions is no different from defining any other kind of functions.

```
# Test whether a list is empty
def is_empty(L):
   return L == []
Is_empty = lambda L: L == []
is_empty([])
True
is_empty(squares)
False

# Test if the first two elements of a list are the same
def same_start(L):
    return L[0] == L[1]
Same_start = lambda L: L[0] == L[1]
```

Extracting sublists using filter

- Filter is used to find elements of a list that satisfy some condition. The condition is given in the form of a boolean function.

```
def non_zero(x):
    return x != 0

non_zero(1)
True
non_zero(0)
False

# Get a list of all the non-zero elements of a list
filter(non_zero, [1, 2, 0, -3, 0, -5])
[1, 2, -3, -5]
```

Using filter

- Let’s write some functions using filter.

```
# Return the elements that are greater than ten
def greater_than_ten(x):
    return x > 10

filter(greater_than_ten, [11, 2, 6, 42])
[11, 42]

# Return all the non-empty lists from a nested list
def is_not_empty(L):
    return L != []

filter(is_not_empty, [[], [1,2,3], [], [4,5,6], []])
[[1, 2, 3], [4, 5, 6]]
```

Exercise 7: filter

1. Define a boolean function increase2 that tests that the first two elements of a list are in increasing order.

```
increase2([2, 5, 4, 9])
True
increase2([2, 2, 5, 0])
False
```

2. Define a function `increasing_lists` that selects from a nested list only those lists that satisfy `increase2`:

```
increasing_lists([[2, 5, 4, 9], [2, 2], [3, 4, 5]])
[[2, 5, 4, 9], [3, 4, 5]]
```

Anonymous functions

- We have been using two notations for defining functions because they will turn out to be useful in different ways. The “def” notation is much more common, and we will see that it has some real advantages. But the “lambda” notation has one big advantage we can use right now:
  - Lambda functions can be used without assigning them to variables - that is, without naming them.

```
map(lambda x: x**2 + x**3, [2,3,4])
[12, 36, 80]
filter(lambda x: x != 0, [1, 2, 0, 3, 0, 6])
[1, 2, 3, 6]
```

- When lambda functions are defined this way, they are called *anonymous functions*. As we use map, filter, and other similar operations more and more, anonymous functions will be really handy.

Exercise 8: Anonymous functions

- Rewrite functions get_second (from exercise 6) and increasing_lists (from exercise 7), using anonymous functions. That is, they should have the form:

```
def get_second(L):
   return map(anonymous_function, L)

def increasing_lists(L):
   return filter(anonymous_boolean_function, L)
```

In each case, you have already defined the function you need in the earlier exercise - it just wasn't anonymous.