NYC DATA SCIENCE ACADEMY

# Get Started with Python

NYC Data Science Academy

# OVERVIEW

- Using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# 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.

# Why Python

in C++

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

in Java

```
public class HelloWorld{

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

in Python

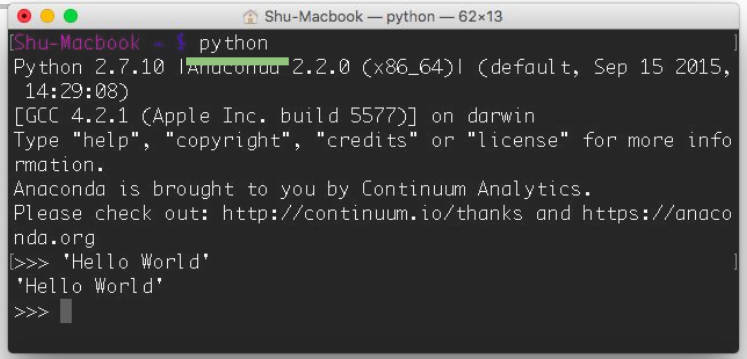
```
print 'Hello World'
```

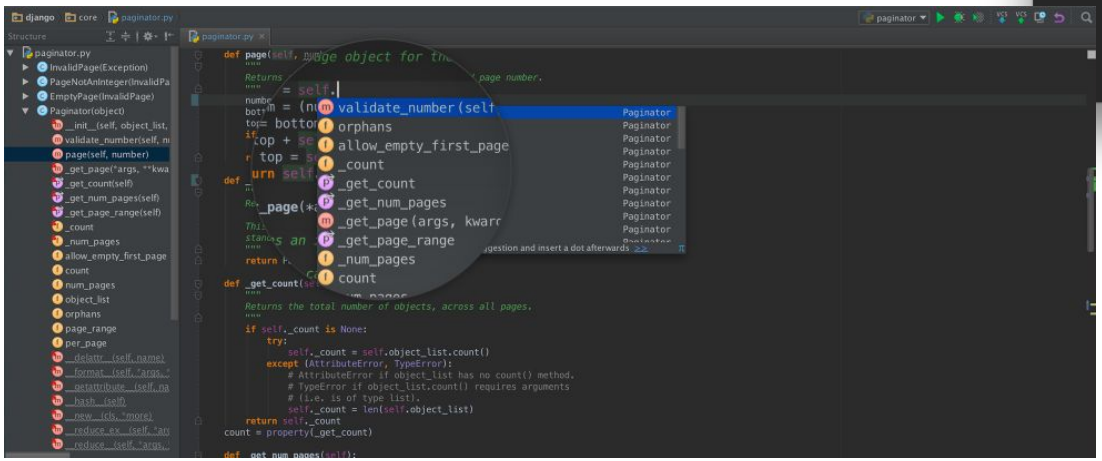
# OVERVIEW

- Using iPython
  - Simple values and expressions
  - Lambda functions and named functions
  - Lists
    - Built-in functions and subscripting
    - Nested lists
  - Functional operators: map and filter
  - List comprehensions
  - Multiple-list operations: map and zip
  - Conditionals
  - Functional operators: reduce

# How to run Python?

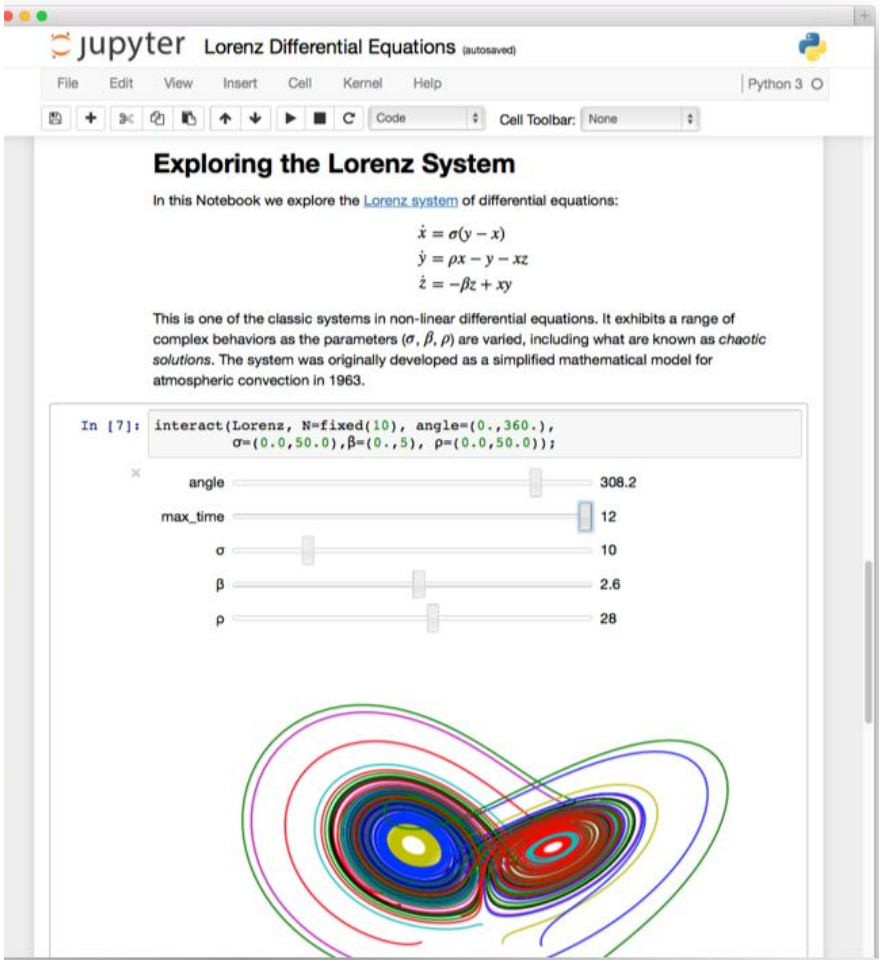
- Python interactive mode. Useful for testing small things.
- Python Scripts (traditional way)
- Using Python IDE (pyCharm, ideal for Productive Python & Web Development).



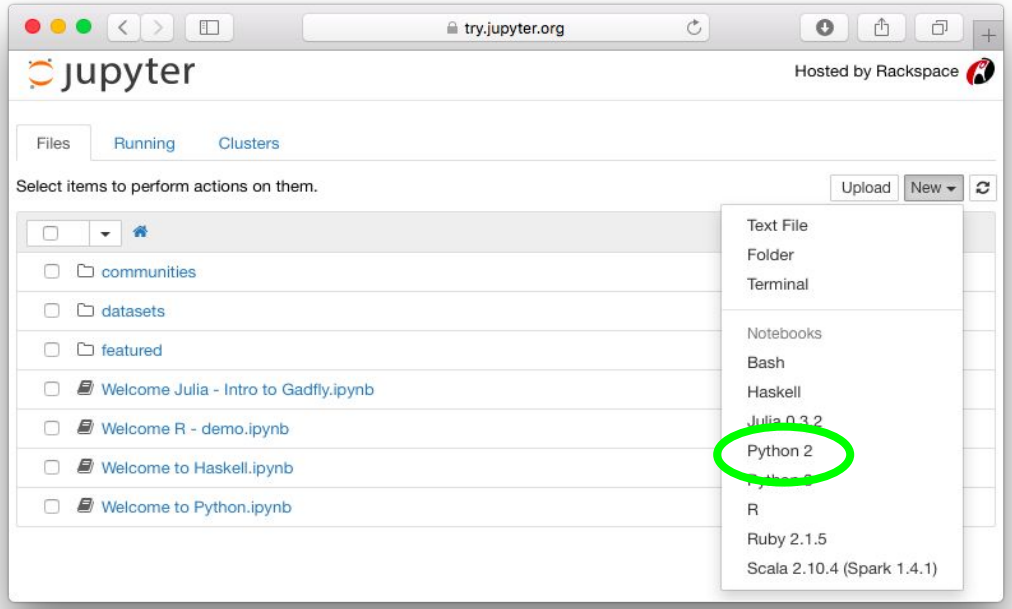
# IPython Notebook

- IPython Notebook: an interactive computational environment
  - Combine code execution, rich text, mathematics, plots and rich media.
- We will use only the basic facilities of IPython: entering, editing, and executing Python code.



# Jupyter and iPython Notebook

- Jupyter is a web app that allows you to create IPython notebooks.



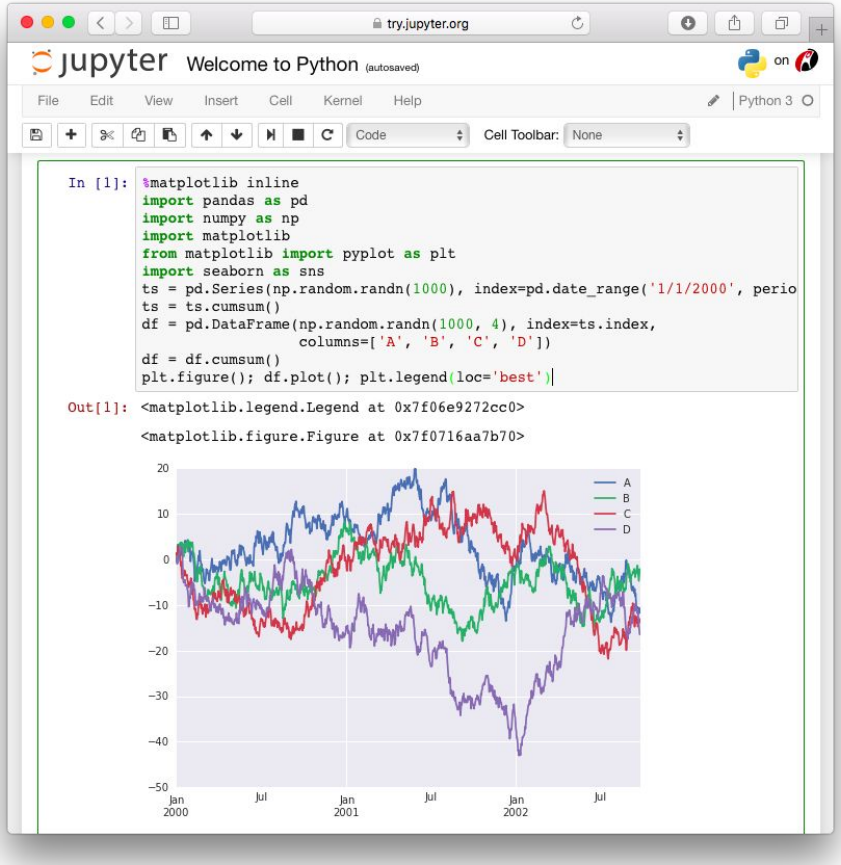

- After starting Jupyter, select Python 2. (Note: Python 2 and Python 3 are not fully compatible; we will be using Python 2 only.)

# Jupyter and iPython Notebooks

- An IPython notebook is a collection of Python code, text, and images displayed in a web browser.
  - An IPython notebook consists of cells. There are several types of cells, including a type where you can enter formatted text. In this class, we will only use *code cells*, in which you enter and execute Python code.
- IPython notebooks also have a persistent representation. IPython notebook files end with `.ipynb`.
- You can either open an existing .ipynb file or create a new one in Jupyter.
- The one which is currently running is shown in green.

# How to use iPython notebook

- Your code goes into the gray box, which is the code cell.
- Click ➤ or press “Shift+Enter” to run the highlighted cell.
- When a cell is running, it is shown as `In [*]`.
- When it’s done, the output is shown as `In [n]`, and a new empty cell is created.
- To stop a running cell, click ■.
- You can go back to a cell by clicking in it, edit the code, and run it again.



# OVERVIEW

- Using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# Simple values and expressions

- Python is used interactively. It can be used as a calculator.


- The figure above shows what you would see in an IPython Notebook. We will put code in a gray textbox; values output by Python will be shown in red. Copy and paste this in your IPython notebook and run it:

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

- *Note*: Comments in Python start with the hash character, #, and extend to the end of the physical line. We will often show comments in italics.

# Simple values and expressions

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

```
1 + 2 * 3      # 7 - * has precedence over +
(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 integer division
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 parens or by adding a \ 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*: \ must be the very last thing on the line (not even followed by spaces), and of course cannot be in a comment.

# 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    # An "assignment statement"
price = 100.50
price * tax
12.5625
```

- When you run an assignment statement, it does not print anything.
- 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:

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

- There is also 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:

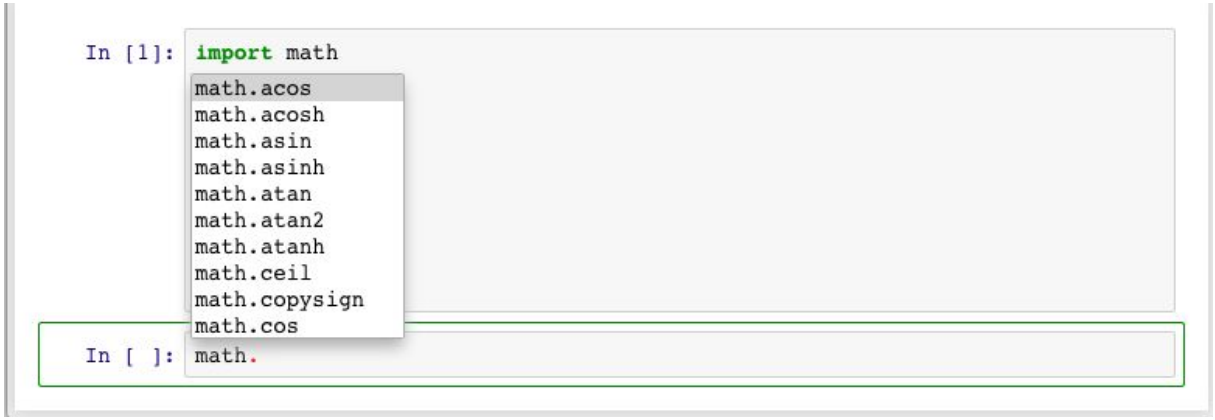
```
import math           # import the math module
math.factorial(5)     # factorial of 5
```

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

```
from math import *
factorial(5)          # no module name
```

# How can I find the functions?

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



# 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 variable names and to 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: `first_index`, `random_nums`
  - One-letter names are used in certain circumstances - e.g. `i`, `j`, `k` when used as indexes - but are otherwise frowned upon.

# Exercise 1: Exploring IPython

- Open an iPython notebook by selecting “python 2” under the “New” dropdown. Rename it to lec_1.
- 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 question 2?

# 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 printed.
- When you assign an expression to a variable, its value is not printed; 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
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# Defining functions

- You have seen how to call functions. Now we'll see how to define them.
- 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 is followed by the function name and the parenthesized list of parameters. The statements in the body of the function start at the next line, and must be indented.
- Call the function in the usual way (after the function has been defined, whether in the same cell or a later 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:

```
def f(x):
    return expression
```

```
f = lambda x: expression
```

  - The first version begins with “def” and has its argument in parentheses. The second version looks like an assignment statement; it uses the keyword “lambda”, and the argument is *not* in parentheses.
  - The first version uses 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(x, y):
    return math.sqrt(x**2 + y**2)

Vector_length = 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 = $\pi r^2$.) 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
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# 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.)
- We will look at lists first, but we'll use strings in our examples. We'll do more with strings in the next class.
- *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.

# 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]
```

*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

- Using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# 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 types of the elements are:
  - [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.

# 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

- Using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# map: Applying functions to all elements of a list

- You will often want to apply a function to all elements of a list. For example, we could read in files and turn them into lists. Applying a function to each element of the list will mean applying the 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
```

# Exercise 6: map

4. Sum up the element in each inner list.

```
sum_each([[1, 2, 3], [4, 5], [6, 7, 8]])
[6, 9, 21]
```

5. Multiply every element by 2.

```
double_each([[1, 2, 3], [4, 5], [6, 7, 8]])
[[2, 4, 6], [8, 10], [12, 14, 16]]
```

# 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 with 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.

# 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
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# List comprehensions

- List comprehensions are another notation for defining lists. They are meant to mimic the mathematical notation of "set comprehensions."
- A list comprehension has the form:

  [ *expression* `for` *x* `in` *list* `if` *x satisfies a condition* ]

  The where clause is optional.

```
[ x * x for x in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]

nested_list1 = [[], ["ab", "cd"], [3, 4, 5]]
[len(l) for l in nested_list1]
[0, 2, 3]

[l[0] for l in nested_list1 if l != []]
['ab', 3]
```

# Exercise 8: List comprehensions

- Write list comprehensions to create the following lists:
  1. The square roots of the numbers in [1, 4, 9, 16]. (Recall that math.sqrt is the square root function.)
  2. The even numbers in a numeric list L. Define several lists L to test your list comprehension. (*n* is even if *n* % 2 == 0.)
- Now define functions that use these list comprehensions:
  3. all_square_roots(L) returns the list of all the square roots of elements of L (which must be numeric). You can use either the def or lambda form; you don't need to use both.
  4. evens(L) returns the even numbers of L. Again, use either def or lambda.

# 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
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# Multiple list operations

- A multiple-list operation is one that combines two lists
  - Add the elements of two lists of the same length
  - Rearrange the elements of one list by using elements of another list as subscripts
  - Select elements of one list corresponding to the True elements in a list of boolean values
- These operations require that we map simultaneously over two lists. There are two ways to do this:
  - Use map. We've used map to map over a single list with a unary function, but it can also be used to map over multiple lists.
  - Use zip to put two lists together, then use unary map.

# Binary map

- map(fun, lis1, lis2) applies fun to pairs of elements from lis1 and lis2
  - lis1 and lis2 must have the same length
  - fun is a binary operation whose arguments are of the correct type.
- If lis1 = [$x_0$, $x_1$, $x_2$, …] and lis2 = [$y_0$, $y_1$, $y_2$, …], then map(fun, lis1, lis2) is equal to [fun($x_0$, $y_0$), fun($x_1$, $y_1$), fun($x_2$, $y_2$), …].
- Examples:

```
map(lambda x, y: x+y, [1,2,3], [10,20,30])
[11, 22, 33]   # [1+10, 2+20, 3+30]
map(lambda x, y: x[y], [[1, 2], [2, 3]], [0, 1])
[1, 3]         # [[1, 2][0], [2, 3][1]]
```

# zip

- zip is a binary function that takes two lists of the same length and makes one list containing pairs of corresponding elements of the two lists.

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8]
zip(a, b)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

- The elements in the zipped list are *tuples*. These are just like lists, but are written using parentheses instead of square brackets. For the most part, the difference between lists and tuples won't matter to you.

- zip provides an alternative to binary map:

```
map(lambda p: p[0]+p[1], zip([1,2,3], [10,20,30]))
[11, 22, 33]
```

# More than two lists

- Both map and zip can actually apply to more than two lists, in the "obvious" way:

```
a = [1, 2, 3]
b = [5, 6, 7]
c = [9, 10, 11]
zip(a, b, c)
[(1, 5, 9), (2, 6, 10), (3, 7, 11)]
```

```
map(lambda t: t[0]+t[1]+t[2], zip(a,b,c))
[15, 18, 21]
map(lambda x, y, z: x+y+z, a, b, c)
[15, 18, 21]
```

# The operator module

- If you want to apply a built-in operation like + in a map, you can use a lambda function - `lambda x, y: x+y` - but there is a simpler way. The `operator` module defines named versions of the built-in operations, just for use in functions like map. For example:

```
from operator import *
map(add, [1,2,3], [4, 5, 6])
map(mul, [1,2,3], [4, 5, 6])
```

- The documentation page https://docs.python.org/2/library/operator.html gives the complete list of operator names; see the table at the bottom of the page.

# Exercise 9: Operating on multiple lists

1. Write a function number_items that adds sequence numbers to each item in a list:

```
number_items(['a', 'b', 7])
[(1, 'a'), (2, 'b'), (3, 7)]
```

   Create the numeric sequence using `range`, and then use zip.

2. Write eq_elts that compares the elements of two equal-length lists for equality:

```
eq_elts([1, 2, 3], [3, 2, 1])
[False, True, False]
```

   Import the operator module, look for the equality operator, and use in the three-argument version of map.

# OVERVIEW

- Using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# Conditionals in functions

- We have seen boolean functions used in the filter operator. They can also be used inside functions, to do different calculations depending upon properties of the input.
- For example, recall the function `firstelt` that returns the first element of a list:

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

It "crashes" if its argument is the empty list. Suppose we would like it to instead return `None` in that case; `None` is a special value in Python that is often used for this kind of thing. We can do that with a conditional:

```
def firstelt(L):
    if L == []:
        return None
    else:
        return L[0]
```

# Conditionals in functions

- The syntax for a conditional in a function is:

```
if condition:                # any boolean expression
    return expression        # return is indented from if
else:                        # else starts in same column as if
    return expression        # return is indented from else
```

- The syntax in lambda definitions is different:

```
lambda x: expression if condition else expression
```

  - For example, here is firstelt in lambda syntax:

```
Firstelt = lambda L: None if L==[] else L[0]
```

# Conditionals in functions

- Conditionals can be nested arbitrarily:
  - Return A if c1 is true, B if c1 is false but c2 is true, and C if both are false:

```
if c1:
   return A
else:
   if c2:
      return B
   else:
      return C
```

# Conditionals in functions

- Having an if follow an else is so common there is special syntax for it:

```
if c1:
   return A
elif c2:
   return B
else:
   return C
```

# Conditionals in functions

- Return A if c1 and c2 are true, B if c1 is true but not c2, C if c1 is false but c3 is true, and D if c2 and c3 are both false:

```
if c1:
   if c2:
      return A
   else:
      return B
elif c3:
   return C
else:
   return D
```

# Conditionals in functions

- The nesting of ifs can be as deep as you want.
- The if and its corresponding else must start at the same column. Nested ifs or returns within the true or false branch must be indented.

```
if condition:
    true branch
else:
    false branch
```

- An elif must be at the same indentation level as its corresponding if; the elif itself has a matching else that must be at the same indentation level.

```
if condition1:
    condition1 true branch
elif condition2:
    condition2 true branch
else:
    false branch
```

# Exercise 10: Conditionals

- Define these functions. (To save time, you can use either syntactic form you wish; you don't need to use both.)

  1. absolute(x) returns the absolute value of x. (Don't use the built-in abs function.)
  2. choose(response, choice1, choice2) returns choice1 if response is the string 'y' or 'yes', and choice2 otherwise.
  3. leap_year(y) returns true if y is divisible by 4, except if it is divisible by 100; but it is still true if y is divisible by 400. Thus, 1940 is a leap year, but 1900 isn't, but 2000 is.
  4. Use filter to define a function leap_years that selects from a list of numbers those that represent leap years.

# OVERVIEW

- Using iPython
- Simple values and expressions
- Lambda functions and named functions
- Lists
  - Built-in functions and subscripting
  - Nested lists
- Functional operators: map and filter
- List comprehensions
- Multiple-list operations: map and zip
- Conditionals
- Functional operators: reduce

# reduce: Combining the elements of a list together

- map and reduce are the best known "functional" operators on lists.
- reduce is used to combine the elements of a list using a binary operator.
- The idea is: If you have a list $L = [x_0, x_1, x_2, \ldots, x_n]$, and you have a binary operator $\oplus$, reduce($\oplus$, L) gives: $x_0 \oplus x_1 \oplus \cdots \oplus x_n$. Thinking of it in terms of a two-argument function f, reduce(f, L) is $f(f(\ldots(f(x_0, x_1), x_2), \ldots), x_n)$. If L has only one element (n=0), then that element is returned ($x_0$). If L is empty, it is an error.
- Another version of reduce has three arguments, and is defined simply as: reduce(f, L, z) = reduce(f, [z] + L), i.e. z is returned if L is empty.

# reduce example: sum

- Perhaps the simplest example is summing the elements of a numeric list:

```
from operator import *

reduce(add, [1, 2, 3, 4])
10                  # 1 + 2 + 3 + 4
```

- The + operator is also the list concatenation operator:

```
reduce(add, [['a'], [], ['b', 'c', 'd'], ['e']])
['a', 'b', 'c', 'd', 'e']   # ['a'] + [] +
                            # ['b', 'c', 'd'] + ['e']
```

## How to use reduce

- Often, using reduce is simplified by considering only *two-element lists*. If L is a list [x, y], then, from the definition of reduce, reduce(*f*, L) = *f*(x,y).
- As an example, we'll write the max function on lists. We already know there is a built-in max function, but for practice we'll pretend it doesn't exist and write it using reduce.
- So the problem is to define a function *f* such that reduce(*f*, L) gives the maximum of L. As advised above, we'll start by looking at two-element lists. Here are some examples:
  - reduce(*f*, [3, 4]) = *f*(3, 4) = 4
  - reduce(*f*, [4, 3]) = *f*(4, 3) = 4

## How to use reduce

- For two-element lists, we get the right answer as long as f is the "max" function on two numbers. That's easy to define, so we write:

```
max2 = lambda x, y: x if x>y else y
max_ = lambda L: reduce(max2, L)
```

- We're sure this will work on two-element lists, but will it work on longer lists? Think about a 3-element list:
  - max_([4, 1, 3]) = max2(max2(4, 1), 3) (from the definition of reduce), which equals max2(4, 3), which equals 4.
- You can try it on arbitrary lists:

```
max_([6,5,4,2,7,3])
7
```

# reduce: Find number of highest magnitude

- Obviously, we could write the min function the same way. Here's another variant: Find the number of highest magnitude, whether positive or negative:

```
max_mag([6,-5,4,2,-7,3])
-7
```

- Again, think about the two-element version. It has to return the number that is greatest *when comparing absolute values*:

```
max_abs = lambda x, y: x if abs(x) > abs(y) else y
max_mag = lambda L: reduce(max_abs, L)
```

# Exercise 11: reduce

1. Define product:

```
product([1,2,3,4])
24      # 1*2*3*4
```

2. Define max_first, which applies to nested lists and gives the element with the greatest first element:

```
max_first([[3, 2], [4, 1, 1], [1, 3]])
[4, 1, 1]
```

# reduce: Sum of squares

- The two-element list method does not always work. The problem here is to find the sum of the squares of a list:

```
sum_squares([2, 3, 4])
29      # 4 + 9 + 16
```

- If we think only about two-element lists, we will write:

```
sum_sq = lambda x, y:  x**2 + y**2
```

*which is wrong*:

```
reduce(sum_sq, [2, 3, 4])
185
```

- Looking back at the definition of reduce, we see that what we're getting is: sum_sq(sum_sq(2, 3), 4) = sum_sq($2^2$ + $3^2$, 4) = sum_sq(13, 4) = $13^2$ + $4^2$ = 185.

# reduce: Sum of squares

- Think about reduce this way: In finding the desired value of our function on a list L = [$x_0$, $x_1$, $x_2$, …, $x_n$], we first find the desired value on the first part of the list, [$x_0$, $x_1$, $x_2$, …, $x_{n-1}$], and then add in the last element.
- So, suppose we have list [2, 3, 4], and we have the sum of squares of [2, 3] - namely, 13 - and now we want to add in the square of 4. The function to use for this is:

```
add_sq = lambda x, y:  x + y**2
```

- This *almost* works:

```
reduce(add_sq, [2, 3, 4])
27     # should be 29
```

- Can you see the problem?

## reduce: Sum of squares

- The problem was that our function did not account for the very first element - it didn't square the 2.
- To see how to fix this, consider this:

```
reduce(add_sq, [0, 2, 3, 4])
29      # Yay!
```

- So, we can fix the problem by using the three-argument version of reduce:

```
sum_squares = lambda L: reduce(add_sq, L, 0)
sum_squares([2, 3, 4])
29
```

# Exercise 12: reduce

1. Define sum_of_lengths in two ways: (a) Using reduce; (b) Using map and then the built-in sum function:

```
sum_of_lengths([[1, 1], [], [2, 2], [3]])
5       # len([1, 1]) + len([]) + ...
```

When defining it with reduce, you will run into the same problem as we did with sum_of_squares, and you should apply the same solution.