BST 261: Data Science II

Lecture 2

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

March 21, 2018

Python Review

Why learn Python?

- Python is a clear and powerful programming language
- Elegant syntax makes programs easy to read
- Large standard library supports many common programming tasks
- Python can be used in two different modes
    - **Interactive mode** makes it easy to experiment with the language
    - **Standard mode** is for running scripts and programs from start to finish
- Python programs are much shorter than equivalent C, C++, or Java programs
- High-level language and data types allow concise complex operations

Which Python to learn?

- In addition to core Python, will also use several Python packages to perform scientific computations
- Instead of installing Python packages one at a time, we will use **Python distribution**
- There are currently several Python distributions available, will use Anaconda Python Distribution
- In October 2015, the distribution for Python 3.4 supported 274 packages
- In October 2016, the distribution for Python 3.5 supported 445 packages
- In March 2017, the distribution for Python 3.6 supports 452 packages
- Anaconda includes development environments Jupyter (formerly, IPython Notebook) and Spyder

Python 2 or Python 3?

- Some like Python 2.x and some like Python 3.x
- What are the differences?
  - Short version: Python 2.x is legacy, Python 3.x is the present and future of the language (www.python.org)
  - The final 2.x version 2.7 release came out in mid-2010 (no further major releases)
  - Python 3.0 was released in 2008, Python 3.1 in 2009, Python 3.2 in 2011, Python 3.3 in 2012, Python 3.4 in 2014, Python 3.5 in 2015, and Python 3.6 in 2016

Python 2 or Python 3?

- Downside: breaking backwards compatibility in 3.x means that some software, especially in-house software in companies, still doesn’t work on 3.x
- Guido van Rossum, the original creator of the Python language, decided to clean up Python 2.x with less regard for backwards compatibility
- All recent standard library improvements only available by default in Python 3.x
- Python 3.x is easier for newcomers to learn and several aspects of the core language are more consistent
- Eliminates many quirks that can trip up beginning programmers learning Python 2 (e.g., division by integers)

Basic Data Types

Introduction

- Python contains many data types as part of the core language, such as numbers
- All data in a Python program is represented by objects or by relations between objects
- The value of some objects can change. Objects whose value can change are said to be **mutable**, whereas objects whose value is unchangeable once they are created are called **immutable**
- Python also contains built-in functions that can be used by all Python programs
- The "Python library" consists of all core elements, such as data types and built-in functions, but the bulk of it consists of modules
- To make use of modules in your code, you first need to import them using the `import` statement

Introduction

```
# print object
print("hello world")

# return the largest item
max(2,9,5)

# return a number rounded to n digits after the decimal point (floating point)
round(12.345, 1)

# return a number rounded to the nearest integer (int)
round(12.345)

# return a new sorted list
sorted([4,2,3,1])

# logical operations
2 < 3 < 4
```

Object Types

- Every piece of data stored in a Python program is an object
- Each object has a **type**, **value**, and **identity**
- For example: `str` (type); `"hello"` (value); `2915232` (identity)
- **Attribute** is a name that is attached to a specific object and has a dot between the two: `object.attribute`
- Some attributes are callable (functions) and others are noncallable (variables)
- Objects are characterized by data attributes and methods
  - A **data attribute** is a value attached to an object
  - A **method** is a function attached to an object that performs some operation on the object
- Object type determines which operations the object supports, i.e., which operations you can perform on it
- An instance is an occurrence of an object, and different instances share the same set of attributes

```python
import numpy as np
x = np.array([2,4,6,8])
y = np.array([1,3,9])
x.mean()
y.mean()
x.shape
y.shape
```

Modules

- Python **modules** are libraries of code
- They are imported using the `import` statement
- Python comes with several modules but you can also write your own modules
- For example, mathematical functions are available in the `math` module

```python
from math import pi, sqrt
import math
math.pi
math.e
math.sqrt(10)
math.sin(math.pi/2)
```

Modules

- As programs grow in size, it will be useful to break them into multiple files for easier maintenance
- Large Python programs are usually organized into modules and then loaded using the `import` statement
- The `import` function does three things:
  - Creates a new namespace for all the objects defined in the source file
  - Executes the code in the module within the newly created namespace
  - Creates a name, which matches the name of the module, within the caller that refers to the module namespace
- Subsequent calls to `import` neither load nor execute the code, but they will bind the module name to the module object created earlier
- A namespace is space or container of names to prevent naming conflicts

```python
import numpy as np
import math
math.sqrt(2) # math.sqrt takes in a single number and returns a number
np.sqrt([2,3]) # np.sqrt takes in either a number or a sequence and returns an
    np.array object
```

Object Types

- To find out the type of an object:

```
name = "Amy Poehler"
```

- To find out a list of all attributes (object.attribute):

```
dir(name)
```

- Note that some of the attributes are callable (methods)
- To find out what different methods do:

```
help(name.lower)   # documentation on the string-function "lower"
help(name.lower()) # documentation on "amy meredith poehler" (!)
```

- Note that the above examples use names (object instances), but one can also use object types

```
help(str)
```

Numbers

- Numbers are one type of object in Python
- Python provides 3 numeric types: integers, floating-point numbers, and complex numbers
- Integers have unlimited precision
- Different numeric types can be freely mixed
- Numbers support all the usual arithmetic operations

```
125 + 25.21
125 + 25
125 * 25
125 ** 25
5 / 2       # normal division
5 // 2      # floor division
1 + _
```

- In the interactive mode, the _ operator contains the result of the last operation (very handy!)

Numbers

- Very commonly need to go beyond built-in functions and operations
- We can import the `math` module for this purpose

```python
import math
math.sqrt(math.pi)
math.sin(math.sqrt(math.pi)/2)
```

- `random` is a useful module when dealing with simple settings that require random sampling of a sequence of objects

```python
import random
random.random()
random.choice([10, 15, 20, 25, 30, 35, 40])
random.choice(["Monday", "Wednesday", "Friday"])
```

Boolean Operations

- **Expression is a combination of objects and operators that computes a value**
- Many expressions involve what is known as the Boolean data type
- The Boolean data type is a data type that has only two values: true and false
- In Python, the Boolean type is `bool`, and it has two values: `True`, `False`
- Compare this to the `int` type which can represent any integer
- Operations involving logic, so-called Boolean operations, take in one or more Boolean objects and return one Boolean object
- There are only 3 Boolean operations (`x` and `y` are Boolean; listed here by ascending priority)

```
x or y

x and y

not x
```

Comparisons

- There are 8 comparison operations in Python
- These are commonly used for numeric types
- Can also be used for other types, like sequences, where comparisons are done element wise
- The result of a comparison is either `True`, `False` (returns a Boolean type)
- All have the same priority (higher than Boolean)
- Here `x` and `y` are any objects for which the comparison operators are defined:

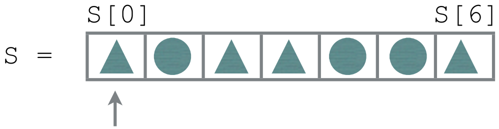
```
x < y
x <= y
x > y
x >= y
x == y
x != y
is
is not
```

- The last two compare object identity and its negation
- We can also chain comparisons in Python

```
2 < 3 < 4
```

Sequences

- In Python, a **sequence** is a collection of objects ordered by **position**
- There are 3 basic sequence types: lists, tuples, and range objects
- Additional sequence types exist for representing strings (text)
- All sequences support **common sequence operations** and each sequence type has its **additional type specific operations**
- These data types are called sequences because the objects they contain form a sequence



- Indexing for sequences starts from zero (0)
- Indexing logic: [from position, to position)

Strings

- **Strings are immutable sequences of characters**
- String literals are enclosed in single (`‘hi’`), double(`"hi"`), or triple quotes (`"""Computer says "No.""""`)
- Examples of common sequence operations on strings

```
S = "Python"
len(S)
S[0]
S[0:6] # slicing
S[-1]
"y" in S # membership
"z" not in S
```

Strings

- **Polymorphism** is the idea that the meaning of an operation, such as + and ∗, depends on the objects being operated on
- This is like in math: each operation has to be defined separately for each type of object

```
S = "Python"
3*5
3*S
3 + 5
S + ' is phun'
print("eight is " + 8)       # string object + integer object (!)
print("eight is " + str(8)) # string object + string object
```

- Here + is referred to as "concatenation" and ∗ as "repetition"

Strings

- The above operations on strings were really generic sequence operations
- Strings are a type of sequence and support all sequence operations
- In addition, strings have operations all their own, which are available as **string methods**
- Methods are functions attached to objects and they are triggered with a call expression

```
S = ‘Python’
S.find(‘y’)
S.replace(‘y’, ‘Y’)
name = ‘Amy Poehler’
name.split(’ ’)
print(name)
name.upper()
print(name)
name.lower()
print(name)
```

Lists

- **Lists are mutable sequences of objects of any type** typically used to store homogeneous items
- Lists are a type of sequence
- Strings vs. lists: Sequences of characters vs. sequences of any objects
- Strings vs. lists: Strings are immutable whereas lists are mutable
- It is common in practice for lists to hold objects of one type only (which may consist of many kinds of nested objects)

```
names = ["Peter", "John", "Mary"]
names[0]
names[1]
names.append("Tom")
names = names + ["Jim"]
import random
random.choice(names)
```

Lists

```python
names.reverse() # reverses the list in place (common mutable sequence operation)
names.sort() # sorts the list i place (list method)
```

- Because lists are mutable, the above method calls actually modify the original list
- Because strings are immutable, they cannot modify the string itself, so many methods return a new string
- Therefore, with strings, if you want to store the result, you need to assign it to a variable

```python
name = "Juan Pedro"
name.upper()
print(name)
name = name.upper()
print(name)
numbers = [1,5,4,7,9,5,3,4,5,6,5,3,4,5,6,7,5,3,2,3,4,5,6,5,3,2,3,1]
numbers.count(4) # common sequence operation
```

Tuples

- **Tuples are immutable sequences** typically used to store heterogeneous data
- Tuples are best viewed as a single object consisting of several parts
- You can use a tuple to return multiple values from a function (coming up)
- Although tuples and lists can often be used to perform similar tasks, creating short lists wastes memory because Python optimizes the performance of certain methods (such as append()) by slightly over-allocating memory

```python
T = (1,2,3,4)
len(T)
T + (5,6)       # concatenation
T[0]
T[1] = 20       # immutable (!)
x = 12.3; y = 22.5
coordinates = (x,y)   # tuple packing
(x, y) = coordinates  # tuple unpacking
x, y = coordinates    # tuple unpacking
coordinates = [(x,x**2) for x in range(10)]
for x, y in coordinates:
    print(x,y)
```

Ranges

- **Ranges are immutable sequences of integers** commonly used in `for` loops

```
list(range(5)) # stop
list(range(1,6)) # start and stop
list(range(0,11,2)) # start stop and step
```

- Can use negative steps but cannot use non-integer steps
- Note that `range` objects are different from `list` objects
- Although could use a list in `for` loops, `range` is much more memory efficient as it stores only start, stop, and step values, calculating individual items as needed

Sets

- **Sets are unordered collections of distinct hashable objects**
- There are two types of sets: `set` is mutable whereas `frozenset` is immutable
- A set is an unordered collection of objects and it cannot be indexed
- The elements of a set are never duplicated, i.e., adding the same element again to a set has no effect (compared to `list.append()`)
- Useful for keeping track of distinct objects and for doing mathematical set operations (union, intersection, set difference)
- Sets come with non-mutating and mutating methods, where the mutating methods can be called only on instances of type `set`

Sets

```python
nodes = set()
nodes = set([2,4,6,8,10,12])
nodes.add(14)
nodes.add(6) # adding a single item at a time
nodes.update([6,14]) # item adding multiple items at a time
nodes.pop() # remove and return an arbitrary element
nodes.remove(4) # remove a specified element
females = set([2,6,8])
males = nodes.difference(females)
print(nodes)                          # difference is a nonmutating method
2 in males                            # testing set membership
```

- There are handy shortcuts for set operations:

```python
males = nodes - females         # set difference
everyone = males | females      # set union
nobody = males & females        # set intersection
```

Sets

- As a simple application, let's use sets to count the number of unique letters in a word

```python
word = "an-ti-dis-es-tab-lish-ment-ar-i-an-ism" # syllabified
word = word.replace("-","")
letters = set(word)
count = len(letters)
```

- There is no need to do this in steps (apart from clarity) if we don't need the intermediate results:

```python
count = len(set("floc-ci-nau-ci-ni-hil-i-pil-i-fi-ca-tion".replace("-","")))
```

Dictionaries

- **Dictionaries are mappings from key objects to value objects**
- Consist of key-value pairs, where keys must be immutable (actually, hashable) and the values can be anything
- Dictionaries themselves are mutable, and can be seen as associative arrays
- Dictionaries can be used for performing fast lookups on unordered data
- Note that dictionaries are not sequences, and therefore do not maintain any type of left-to-right order, i.e., keys and values are iterated over in an arbitrary order
- Diagram of key-value pairs and lack of order

Dictionaries

- Some examples of how to use dictionaries

```
md = {}
md = dict()
age = {‘Tim’: 29, ‘Jim’: 31, ‘Pam’: 27, ‘Sam’: 35}
age[‘Tim’]
age[‘Tim’] = age[’Tim’] + 1
age[‘Tim’] += 1
age[‘Tom’] = 45
```

Dictionaries

- The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are **view objects**
- Provide a dynamic view on the dictionary's entries: when the dictionary changes, the view reflects these changes

```
age = {‘Tim’: 29, ‘Jim’: 31, ‘Pam’: 27, ‘Sam’: 35}
names = age.keys()
age[‘Tom’] = 45
print(names)
‘Tim’ in age
```

- To get a new view of the dictionary's keys

```
age.keys()
```

- To get a new view of the dictionary's values

```
age.values()
```

- Dictionary views support iteration and membership tests

```
‘‘Tom" in names
age[‘Nick’] = 52
‘‘Nick" in names
```

Dynamic Typing

Dynamic Typing

- Some languages are statically typed (e.g., C and C++) and others are dynamically typed (e.g., Python)
- Static typing means that type checking is performed during compile-time
- Dynamic typing means that type checking is performed at run-time
- What is type checking? The process of verifying and enforcing the constraints of types. If you move data from one variable to another, if the types of these variables do not match, you could potentially lose information. Typing, or assigning data types, refers to the set of rules that the language uses to ensure that the part of the program receiving some data knows how to correctly interpret that data.

Dynamic Typing

- Consider the following simple example:

```
x = 3
```

- How does Python know that `x` should stand for an integer?
- Three important concepts: variable (name), object, and reference
- When assigning variables to objects, the following three things happen:

Object: Python creates an object to represent the value 3

Variable: Python creates the variable `x` (if it does not exist yet)

Reference: Python links the variable to the object

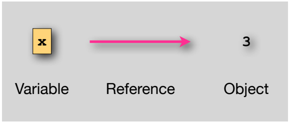


Figure: Schematic of dynamic typing.

Dynamic Typing

- Variables and objects are stored in different parts of computer's memory
- **Variables (names) always link to objects, never to other variables**
- A variable thus becomes a reference to the given object, i.e., a name (possibly one out of many) attached to the object
- Larger objects may link to other objects
- Only objects have types:

```python
x = 3
type(x)
x = "hello"
type(x)
```

- The variable x now references a different type of object (string, not integer)
- However, the type of x itself has not changed because x (the variable) has no type

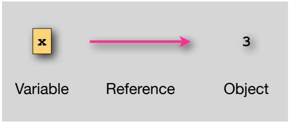


Figure: Schematic of dynamic typing.

Dynamic Typing

- When assigning objects, it's useful to keep in mind the distinction between mutable and immutable objects
  - **Mutable objects** can be altered (lists, dictionaries)
  - **Immutable objects** cannot be altered (numbers, strings)
- Let's start with an easy example:

```
x = 3
y = x
```

- This may be confusing sometimes:

```
x = 3
y = x
x = x - 1
```

- What is the value of `x` and `y` after the last operation?

Dynamic Typing

```
x = 3
y = x
x = x - 1
```

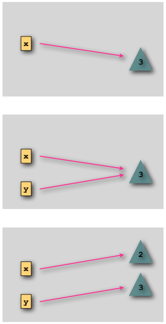


Figure: Schematic of the problem.

- The key to understanding this is to realize that **numbers are immutable objects**
- Consequently, the subtraction results in a **new object** being created
- For **immutable objects** assignment effectively creates a copy

Dynamic Typing

- How about this example for a list?
- The behavior for **mutable** objects (e.g., lists, dictionaries) looks different, although it really follows the same logic

```
L1 = [2,3,4]
L2 = L1
L1[0] = 24
L1
L2
```

Dynamic Typing

```
L1 = [2,3,4]
L2 = L1
L1[0] = 24
L1
L2
```

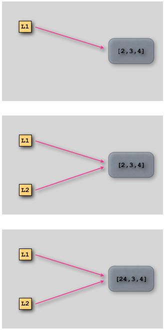


Figure: Schematic of the problem.

- Because the target object can be modified (lists are mutable), the operation does not result in the creation of a new object (compared to numbers)

Dynamic Typing

- Each object has a **type**, **value**, and **identity**
- Mutable objects can be identical in content, and yet be different objects

```
L = [1,2,3]
M = [1,2,3]
L == M
```

```
True
```

```
L is M      # testing for object identity
```

```
False
```

- We can use the `id` function to obtain the identity of an object, which corresponds to its location in memory

```
id(L)
```

```
4316337288
```

```
id(M)
```

```
4312749840
```

```
id(L) == id(M)
```

```
False
```

Dynamic Typing

- What if you wanted to copy a list, not just another reference (synonym) for it?
- The most elegant way:

```python
L = [1,2,3,4,5,6,7]
M = list(L)
M is L
```

```
False
```

- Could also use slicing:

```python
M = L[:]
```

- For more complex structures, there is also the `copy` module:

```python
import copy
newlist = copy.copy(L)
newlist = copy.deepcopy(L)
```

- A **shallow copy** constructs a new compound object and then **inserts references** into it to the original
- A **deep copy** constructs a new compound object and then, recursively, **inserts copies** into it of the original objects

Compound Statements

Python Compound Statements

- Compound statements contain (groups of) other statements and they affect or control the execution of those other statements in some way
- Compound statements typically span multiple lines
- A compound statement consists of one or more **clauses** where a clause consists of a header and a block or suite of code
- The clause headers of a particular compound statement start with a keyword, end with a colon, and are all at the same indentation level
- **A block or suite of code of each clause however must be indented to indicate that it forms a group of statements that logically fall under the header**
- There are no hard-and-fast rules about indentation as long as you are consistent
- Tab and 4 spaces are probably the most common choices
- Here's an example of a compound statement with 1 clause (1 header line + 2 lines of code in the block)

```python
if x > y:
    difference = x - y
    print("x is greater than y")
print("But this gets printed no matter what!")
```

Python if Statement

- The Python `if` statement:

```
if test:
    [block of code]
elif test:
    [block of code]
else:
    [block of code]
```

- The `if` statement selects from among one or more actions, and it **runs the block associated with the first `if` or `elif` test that is true**, or the `else` suite if all are false
- This example computes the absolute value of a difference, i.e., the distance between the numbers x and y
- Python has the built-in `abs` function for this purpose, so this is just a pedagogical example

```
if x > y:
    absval = x - y
elif y > x:
    absval = y - x
else:
    absval = 0
```

Python for Statement

- The Python `for` statement:

```
for target in sequence:
    [block of code]
```

- The `for` loop is a **sequence iteration that assign items in** `sequence` **to** `target` **one at a time** and runs the block of code for each item
- Unless the loop is terminated early with the `break` statement, the block of code is run as many times as there are items in the sequence

Python for Statement

```python
names = ['Peter', 'John', 'Mary', 'Helen', 'Tom', 'Nicholas']
for name in names:
    print(name)
for x in [0,1,2,3,4,5,6,7,8,9,10]:
    print(x)
for x in range(11):
    print(x)
```

- There is also an optional `else` clause to the `for` loop, which is executed when the loop terminates through exhaustion of the sequence, but not when the loop is terminated by a `break` statement

Python for Statement & Dictionaries

- Dictionaries are often accessed and manipulated using the the `for` statement

```
age = {'Tim': 29, 'Jim': 31, 'Pam': 27, 'Sam': 35}
age.keys()
age.values()
for person in age:
    print(person, age[person])
for person in age.keys():       # another way to achieve the same
    print(person, age[person])
for person in sorted(age.keys()):
    print(person, age[person])
for person in sorted(age.keys(), reverse=True):
    print(person, age[person])
```

Python for Statement & List Comprehensions

- A common operation is to take an existing list, apply some operation to all of the items of the list, and create a new list containing the results
- In Python there is an operator for this task known as a **list comprehension**
- Consider the following approach to computing squares of a list of numbers:

```python
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []
for n in numbers:
    squares.append(n*n)
```

- This can be easily implemented as a list comprehension:

```python
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [n*n for n in numbers]
```

- If we just care about the squares, we can do the following:

```python
squares = [n*n for n in range(0,11)]
```

- The general syntax is as follows:

```python
[expression for item_1 in iterable_1 if condition_1
            for item_2 in iterable_2 if condition_2
...
            for item_N in iterable_N if condition_N]
```

Python for Statement & Handling Files

- We commonly want to read data from a file or write data to a file
- The syntax for reading a file line-by-line:

```
for line in open('input.txt'):
    line = line.rstrip()
    line = line.split(',')
```

- Or more economically:

```
for line in open('input.txt'):
    line = line.rstrip().split(',')
```

- Why won't the following work?

```
for line in open(''input.txt"):
    line = line.split(',').rstrip()      # this will not work (!)
```

- Writing a file line-by-line:

```
F = open('output.txt', 'w')
F.write('Hello there! \n')    # need the newline character here
F.close()
```

- The default setting is that we are opening a file for reading, but this can also be made explicit:

```
F.open('nput.txt', 'r')
```

Python Statements

- The Python `while` is used for repeated execution as long as an expression is true:

```
while expression:
    [block of code]
else:
    [block of code]
```

- The while loop repeatedly tests the expression
- If the expression is true, it executes the first block of code
- If the expression is false, it executes the second block of code, if present, and terminates the loop
- Use of `for` and `while` depends on whether looping over a known sequence or whether looping while some condition is true

Python Functions

Python Statements

- Statements specify what the program should do
- End of line terminates a statement (no semicolons etc.)
- We can for example read input from the user and then print it out:

```python
name = raw_input("Enter your name: ")
print("Hello " + name + "!")
```

- Backslash can be used to continue a (long) statement to the next line:

```python
print("This is a very long line so it is very nice to be able to \
  continue entering it on the next line.")
```

Python Functions

- Functions are devices for grouping statements so that they can be easily run more than once in a program
- Functions maximize code reuse and minimize code redundancy
- Functions enable dividing larger tasks into smaller chunks (procedural decomposition)
- Functions are written using the `def` statement
- The `return` statement sends the result object back to the caller

```python
def add(a,b):
    result = a + b
    return result
...
my_sum = add(2,3)
```

Python Functions

- All names assigned in a function are local to that function and exist only while the function runs (unless using the `global` statement)
- Arguments are matched by position
- Use **tuples** to return multiple values from a function:

```python
def add_and_subtract(a,b):
    sm = a + b
    df = a - b
    return (sm, df)        # parenthesis are optional here
...
my_sum, my_diff = add_and_subtract(5, 18)
```

Python Functions

- Functions do not exist until Python reaches and runs the `def` statement
- A function is not executed until the given function is called using the `function()` syntax
- The `def` statement creates an object and assigns it to a name

```
type(add)
```

```
<type ‘function’>
```

```
newadd = add
type(newadd)
```

```
<type ‘function’>
```

```
newadd(4,5)
```

Python Functions

- Arguments are passed by assigning objects to local names (object reference)
    - Mutable arguments are passed “by pointer” such that the original object gets modified
    - Immutable arguments are passed “by value” because it’s not possible to modify an immutable object so that a new object is created

```python
def modify(mylist):
    mylist[0] *= 10
...
L = [1,3,5,7,9]
modify(L)
print(L)
```

```
[10, 3, 5, 7, 9]
```

Python Functions

- Let's look at a slightly more complex example
- This function returns the intersection of two sequences

```python
def intersect(seq1,seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
...
A = [1,3,5,7,9]
B = [2,3,4,5,6]
C = intersect(A,B)
```

Python Functions

- Remember polymorphism? The meaning of an operation depends on the type of the objects being operated upon
- For integers, the + operator means addition
- For strings, the + operator means concatenation

```python
def add(a,b):
    return a + b
...
add(12,23)
add('Peter', 'John')
```

Example: Password Generator

- Let’s generate a simple program to generate passwords
- Would like to be able to specify the character set and password length
- `random.sample` is used for random sampling **without replacement**

```python
import random

def password(length):
    characters = "abcdefghijklmnopqrstuvwxyz"
    pw = str()
    for i in range(length):
        pw = pw + random.choice(characters)
    return pw
```

Understanding Common Errors

Understanding Common Errors

- Not reading and/or understanding common error messages
- Not understanding basic built-in types (e.g., dictionaries have no internal ordering)
- Trying to do an operation that is not supported by the object
- Accessing the object in a wrong way (e.g., key vs. index)

Understanding Common Errors

- Trying to modify immutable objects
- Not understanding nested objects
- Not knowing the type of an object
- Trying to operate on two objects of different type (e.g., trying to concatenate a string and a number)

NumPy Basics

Overview of NumPy

- NumPy is a Python module designed for scientific computation
- NumPy has several very useful features
  - NumPy arrays, which are N-dimensional array objects, are a core component of scientific and numerical computation in Python
  - NumPy provides tools for integrating C, C++ and Fortran code in your Python code
  - NumPy also provides many useful tools to help you perform linear algebra, generate random numbers, and much much more.
- More info about NumPy at `http://www.numpy.org/`
- We will always use the following way to import NumPy:

```
import numpy as np
```

NumPy Basics

- NumPy arrays are an additional data type provided by NumPy and they are used for representing vectors and matrices
- Unlike dynamically growing Python lists, NumPy arrays have a size that is fixed when they are constructed
- Elements of NumPy arrays are also all of the same data type (by default floating point numbers), leading to more efficient and simpler code than using Python's standard data types
- Let's start by constructing an empty vector and an empty matrix (note the type of input argument)

```
zero_vector = np.zeros(5)
zero_matrix = np.zeros((5,3))
```

NumPy Basics

- You can also construct arrays of 1s using the `np.ones` function, and its syntax is identical to the syntax of the zeros function
- You can create an empty array using using the `np.empty` function, which allocates the requested space for the array but does not initialize it (fast but use with care)
- We can also construct NumPy arrays using specified values, in which case we use the `np.array` function and the the input is a sequence of numbers, typically a list

```
x = np.array([1,2,3])              # Contructed from a list
y = np.array([2,4,6])
X = np.array([[1,2,3], [4,5,6]])   # Contructed from nested lists (rows)
Y = np.array([[2,4,6], [8,10,12]])
A = np.array([[2,4], [6,8]])
B = np.array([[1,3], [5,7]])
```

NumPy Basics

- A 2-dimensional NumPy array is constructed by specifying the elements of each row as its own list, and the entire table consists of a list of these lists (rows)

```
np.array([[1,3], [5,9]])
```

- We can transpose an array using the `transpose` method

```
A = np.array([[2,4], [6,8]])
C = A.transpose()
C = A.T
```

NumPy Basics

- It is easy to index and slice NumPy arrays regardless of their dimensions
- With 1-dimensional arrays, we can index a given element by its position, keeping in mind that indices start at zero (0)
- With 2-dimensional arrays, the first index specifies the row of the array and the second index specifies the column of the array
- We can also slice NumPy arrays by specifying the start index and the stop index, but remember that Python stops before hitting the stop index
- With multidimensional arrays, you can use the : (colon) character in place of a fixed value for an index, which means that array elements corresponding to all values of that particular index will be returned
- For a 2-dimensional array, using just 1 index returns the given row, which is consistent with the construction of 2d arrays as lists, the rows of the array, nested with a list

NumPy Basics

- Some examples of using NumPy indices

```
x[2]
x[0:2]
z = x + y
x[0:2] + y[0:2]
X[0,:] + Y[0,:] # row 0 (and all columns) of X + row 0 (and all columns) of Y
X[0] + Y[0] # access elements of 2D array using single indices
X[:,1] + Y[:,1] # column 1 (and all rows) of X + column 1 (and all rows) of Y
```

- It's important to keep operator polymorphism in mind, i.e., the concept that the behavior of any operation, such as the plus sign placed between two objects, depends on the type of those two objects

```
[2,4] + [6,8]
np.array([2,4]) + np.array([6,8])
```

NumPy Basics

- Numpy arrays can also be indexed with other arrays or other sequence-like objects, such as lists

```
z1 = np.array([1,3,5,7,11,13])
z2 = z1 + 1
ind = [0,2,4]
z1[ind]
ind = np.array([0,2,4])
z1[ind]
```

NumPy Basics

- NumPy arrays can also be index using logical indices, arrays consisting of the Boolean elements True and False

```
z1 > 6
```

- We can now use the Boolean array, also called a logical array, to index another vector, which returns those elements of the array for which the corresponding value in the Boolean vector is True

```
z1[z1>6]
z2[z1>6]
ind = z1>6 #explicit construction of a logical index vector
z1[ind]
z2[ind]
```

NumPy Basics

- When you slice an array using the `:` operator, you get a view of the object, meaning that if you modify the slice, the original array will also be modified

```python
# Use slicing to get a view to an array.
z1 = np.array([1,3,5,7,11,13]) #redef
w = z1[0:3]
w
w[0] = 3
w
z1
```

NumPy Basics

- This is in contrast with what happens when you index an array, in which case a copy of the original data is returned

```
# Use indexing to get a copy of an array.
z1 = np.array([1,3,5,7,11,13]) #redef
ind = np.array([0,1,2])
w = z1[ind]
w
w[0] = 3
w
z1
```

- **In summary, indexing gives a copy, slicing gives a view**

NumPy Basics

- NumPy provides different ways to construct arrays with fixed start and end values, such that the other elements are uniformly spaced between them
- To construct an array of 10 linearly spaced elements starting with 0 and ending with 100 (end point included)

```
np.linspace(0, 100, 10)
```

- To construct an array of 10 logarithmically spaced elements between 10^1 and 10^2 (end point included)

```
np.logspace(1, 2, 10)
```

- In the above `logspace`, the sequence starts with the base value, usually 10, raised to the power specified by the first argument, and it ends with the base value raised to the power given by the second argument
- To construct an array of 10 logarithmically spaced elements between 250 and 500, we can do the following

```
np.logspace(np.log10(250), np.log10(500), 10)
```

NumPy Basics

- You can check the shape of an array with (no parentheses)

```
X.shape
```

- You can check the number of elements of an array with (no parentheses)

```
X.size
```

- Sometimes we need to examine whether any or all elements of an array fulfill some logical condition
- Let's generate a small 1D array and check if
    - Any of the entries are greater than 0.9
    - All of the entries are greater than or equal to 0.1

```
x = np.random.random(10)
np.any(x > 0.9)
np.all(x >= 0.1)
```

- Note that the output is either `True` or `False` for the whole array:
    - Either there is or is not one or more entries that are greater than 0.9
    - Either all entries are greater than or equal to 0.1 or they are not

Linear Algebra with NumPy

- NumPy has a large collection of tools for doing linear algebra
- Dot product of two arrays, which may be vectors (inner product) or matrices (matrix product)

```
a = np.array([1,2,3])
b = np.array([2,4,6])
a @ b
```

```
28
```

- Compute the (multiplicative) inverse of a matrix

```
X = np.array([[0, 1, 2], [1, 0, 3], [4, -3, 8]])
X_inv = np.linalg.inv(X)
X @ X_inv
```

```
array([[ 1., 0., 0.], [ 0., 1., 0.], [ 0., 0., 1.]])
```

Linear Algebra with NumPy

- We can solve a linear system of equations

```
A = np.array([[1, -2, 1], [0, 2, -8], [-4, 5, 9]])
b = np.array([0, 8, -9])
x = np.linalg.solve(A,b)
```

```
array([ 29., 16., 3.])
```

- Compute the eigenvalues of a matrix

```
A = np.array([[3, -2], [1, 0]])
np.linalg.eigvals(A)
```

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

- Compute the eigenvalues and right eigenvectors of a square array (matrix)

```
eigenvalues, eigenvectors = np.linalg.eig(A)
```

Linear Algebra with NumPy

- Note that the eigenvectors are column vectors in the returned matrix

```
eigenvectors[:,0] # first eigenvector
eigenvectors[:,1] # second eigenvector
```

- We can compute each side of the equation $\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x}$

```
A @ eigenvectors[:,1] # LHS
eigenvalues[1] * eigenvectors[:,1] # RHS
```

Linear Algebra with NumPy

- In simple linear regression the goal is to predict a quantitative response y on the basis of a single predictor variable x
- It assumes the following relationship between the random variables x and y

$$\mathrm{y} = \beta_0 + \beta_1 \mathrm{x} + \epsilon \tag{1}$$

- Here ϵ is mean-zero random error term
- Once we have used training data to produce estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ for the model coefficients, we can predict future values of y:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x \tag{2}$$

- Here $\hat{y}$ indicates a prediction (a specific value) of the random variable y on the basis of $\mathrm{x} = x$ (a specific value)

Linear Algebra with NumPy

- The most common approach to estimating the model coefficients β_0 and β_1 involves minimizing the least squares criterion
- We define the ith residual as $e^{(i)} = y^{(i)} - \hat{y}^{(i)}$, the difference between the ith observed response value and the ith response value predicted by the model
- The residual sum of squares (RSS) is defined as

$$(y^{(1)} - \hat{\beta}_0 - \hat{\beta}_1 x^{(1)})^2 + \cdots + (y^{(n)} - \hat{\beta}_0 - \hat{\beta}_1 x^{(n)})^2 \tag{3}$$

- Using calculus, one can show that the minimizers are

$$\begin{aligned} \hat{\beta}_1 &= \frac{\sum_{i=1}^{n} (x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^{n} (x^{(i)} - \bar{x})^2} \\ \hat{\beta}_0 &= \bar{y} - \hat{\beta}_1 \bar{x} \end{aligned} \tag{4}$$

- Here $\bar{x}$ and $\bar{y}$ are the corresponding sample means

Linear Algebra with NumPy

- In multiple linear regression the goal is to predict a quantitative (scalar) response y on the basis of several predictor variables, and the model takes the following form:

$$\mathrm{y} = \beta_0 + \beta_1 \mathrm{x}_1 + \cdots + \beta_p \mathrm{x}_p + \epsilon \tag{5}$$

- Including a constant as one of the predictors, and setting $\mathbf{x} = [\mathrm{x}_1, \ldots, \mathrm{x}_p]^{\mathrm{T}}$ and $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_p]^{\mathrm{T}}$, we can write the above in matrix form as

$$\mathrm{y} = \mathbf{x}^{\mathrm{T}} \boldsymbol{\beta} + \epsilon \tag{6}$$

- For the ith observation this becomes

$$y^{(i)} = \boldsymbol{x}^{(i)\mathrm{T}} \boldsymbol{\beta} + \epsilon^{(i)} \tag{7}$$

- For a total of n observations, we can stack the equations together to give

$$\boldsymbol{y} = \boldsymbol{X} \boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{8}$$

- Here $\boldsymbol{y}$ is a column vector of the n responses and $\boldsymbol{X}$ is an $n \times p$ matrix of predictors, where each row of the matrix correspond to one observation

Linear Algebra with NumPy

- We can now write down the least squares criterion

$$\begin{aligned} L(\boldsymbol{\beta}) &= \|\boldsymbol{y}-\boldsymbol{\mu}\|^2 = \|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\|^2 = (\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}) \\ &= \boldsymbol{y}^{\mathrm{T}}\boldsymbol{y} - 2\boldsymbol{y}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta} \end{aligned} \tag{9}$$

- Recall the partial derivatives of a linear form and a quadratic form

$$\begin{aligned} \frac{\partial(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{x})}{\partial\boldsymbol{x}} &= \boldsymbol{a} \\ \frac{\partial(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x})}{\partial\boldsymbol{x}} &= (\boldsymbol{A}+\boldsymbol{A}^{\mathrm{T}})\boldsymbol{x} \end{aligned} \tag{10}$$

- We now obtain the OLS solution by differentiating $L(\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$ and setting the derivative to zero

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} \tag{11}$$

Linear Algebra with NumPy

- We can compute the least-squares solution to a linear matrix equation

```
np.random.seed(1)
x = np.linspace(0,10,15)
y = 1 + 2*x + np.random.normal(loc=0, scale=1, size=x.shape)
```

- We can stack a vector of ones and $\boldsymbol{x}$ to create a design matrix X

```
X = np.vstack([np.ones(len(x)), x]).T
beta = np.linalg.lstsq(X, y)[0] # coefficients only
```

```
array([ 0.93683383, 1.99740465])
```

- We can also compute the OLS solution using matrix operations:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} \tag{12}$$

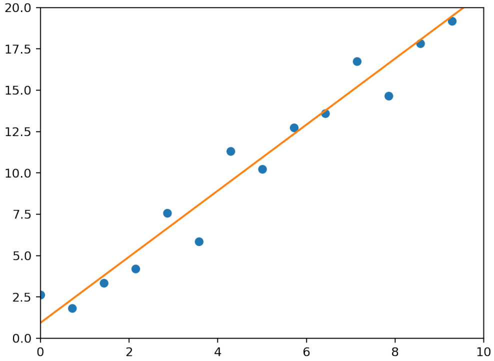
```
from numpy.linalg import inv
beta = inv(X.T @ X) @ X.T @ y
```

```
array([ 0.93683383, 1.99740465])
```

Linear Algebra with NumPy

- Let's plot the result

```
import matplotlib.pyplot as plt
%matplotlib notebook
plt.plot(x, y, "o")
plt.plot(x, X @ beta)
plt.axis([0, 10, 0, 20]);
```


20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
2
4
6
8
10