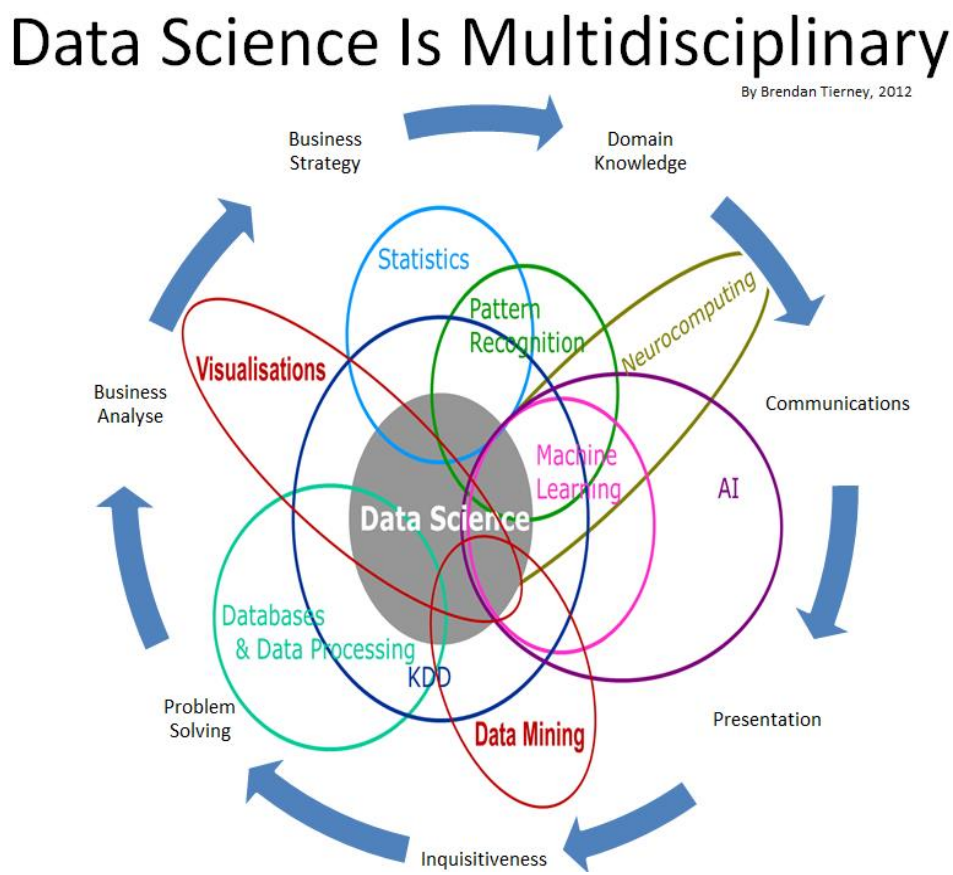
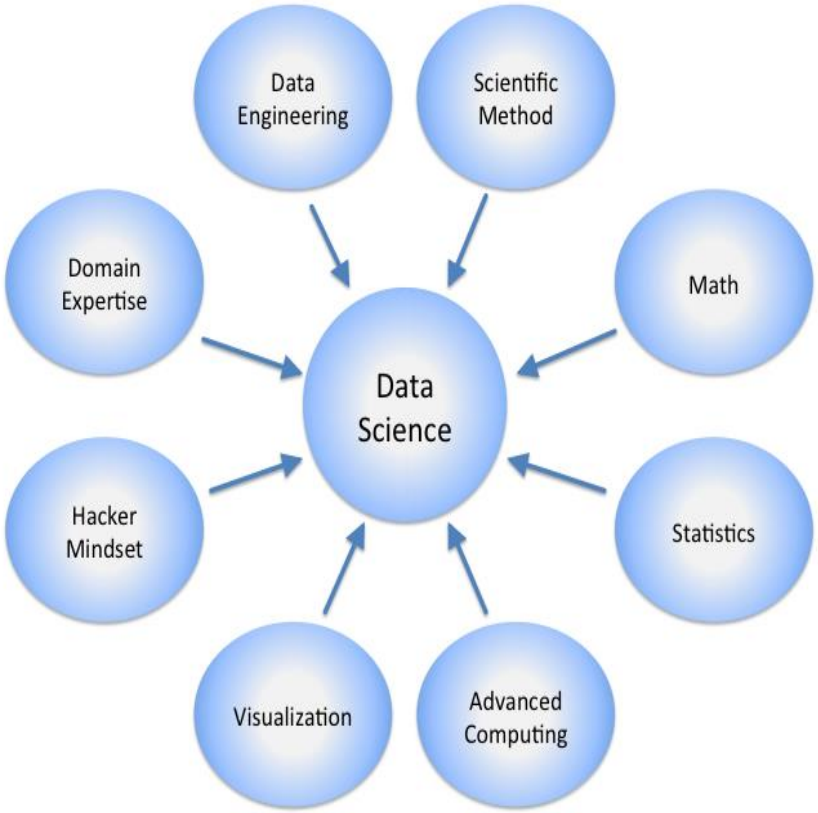
PYTHON PROGRAMMING 3

Anasua Sarkar
Computer Science & Engineering Department
Jadavpur University
anasua.sarkar@jadavpuruniversity.in

Data Science – A Definition

Data Science is the science which uses computer science, statistics and machine learning, visualization and human-computer interactions to collect, clean, integrate, analyze, visualize, interact with data to create data products.





Python : Introduction

- Most recent popular (scripting/extension) language
  - although origin ~1991
- heritage: teaching language (ABC)
  - Tcl: shell
  - perl: string (regex) processing
- object-oriented
- It includes modules for creating graphical user interfaces, connecting to relational databases, generating pseudorandom numbers, arithmetic with arbitrary-precision decimals, manipulating regular expressions, and unit testing.
- Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram and Spotify. The social news networking site Reddit is written entirely in Python.

What's Python?

- Python is a general-purpose, interpreted high-level programming language.
- Its syntax is clear and emphasize readability.
- Python has a large and comprehensive standard library.
- Python supports multiple programming paradigms, primarily but not limited to object-oriented, imperative and, to a lesser extent, functional programming styles.
- It features a fully dynamic type system and automatic memory management

@Yang Li

Python structure

- modules: Python source files
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

High-level data types

- Numbers: int, long, float, complex
- Strings: immutable
- Lists and dictionaries: containers
- Other types for e.g. binary data, regular expressions, introspection
- Extension modules can define new “built-in” data types

Simple data types

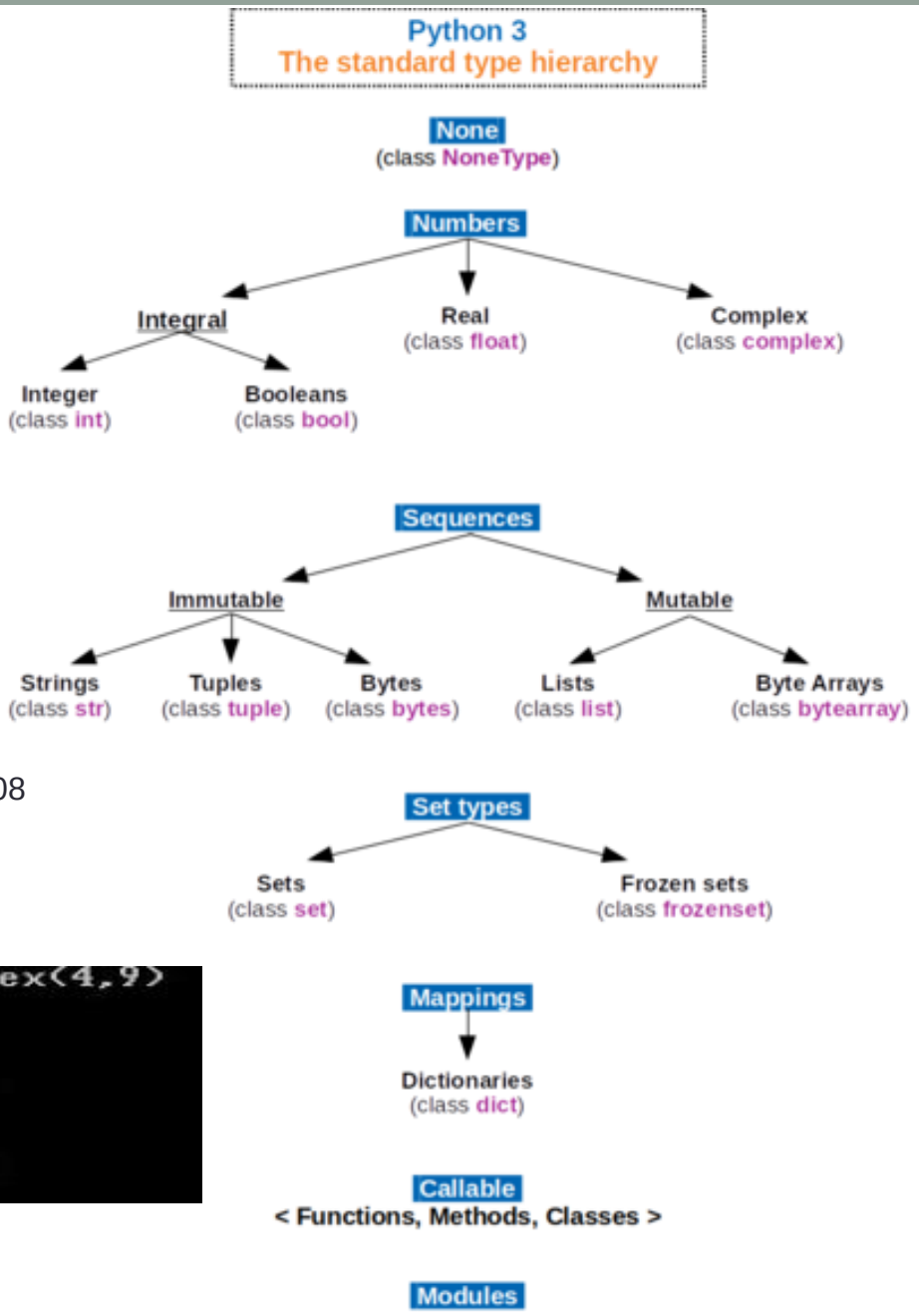
- Numbers
- integers,
  - int : from -2^{31} to $(2^{31}-1)$
- long integers - interpreter will add L

```
>>> 2147683779765
2147683779765L
>>> 2635478778644*1000
2635478778644000L
```

- floating point numbers,
  - ranges approximately from -10 to 10^308
  - 16 digits of precision
- complex numbers - `1j+3, abs(z)`

```
>>> complex_num1=complex(4,9)
>>>
>>> complex_num2=5+9j
>>>
>>> print complex_num1
(4+9j)
>>>
>>> print complex_num2
(5+9j)
```

- Booleans are **`False`** or **`True`**



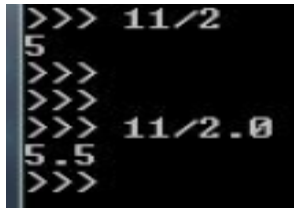
Numbers

- The usual notations and operators
  - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0 # float(1)/2 -> 0.5
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
  - 1j**2 -> (-1+0j)
- Boolean - 'True' or 'False'

Arithmetic Expression

- **Bracket, Of, Division, Multiplication, Addition, and Subtraction** (**BODMAS**)
- The decreasing precedence order is as follows:
  - Exponent
  - Unary negation
  - Multiplication, division, modulus
  - Addition, subtraction
- Addition of two int data types can produce a long integer.
- **Mixed mode conversion**
- int(), float()

Operator
**
*
/
%
+
–



Numeric Types

- Integer
- Boolean
- Real - Decimal type
- Fraction type
- Complex

Built-in Types

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

Complex Numbers

- Complex numbers are a distinct core object type in Python.

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
>>> oct(64), hex(64), bin(64)
('0100', '0x40', '0b1000000')
```

```
>>> 0o1, 0o20, 0o377 # Octal literals
(1, 16, 255)
>>> 0x01, 0x10, 0xFF # Hex literals
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111 # Binary literals
(1, 16, 255)
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
```

Other Numeric Types

Decimal Type

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
>>> 0.1 + 0.1 + 0.1 - 0.3 # This should be zero (close, but not exact)
5.551115123125783e-17
>>> Fraction.from_float(1.75) # Convert float -> fraction: other way
Fraction(7, 4)
```

Fraction Type

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3) # Numerator, denominator
>>> y = Fraction(4, 6) # Simplified to 2, 3 by gcd
>>> print(y)
2/3
>>> x * y
Fraction(2, 9)
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)
```

Built in constants

A small number of constants live in the built-in namespace. They are:

`False`

The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

`True`

The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

`None`

The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`.

Sequences

- Sequences allow you to store multiple values in an organized and efficient fashion.
- There are seven sequence types:
  - strings,
  - Unicode strings,
  - lists,
  - tuples,
  - bytearrays,
  - buffers, and
  - xrange objects.
- Dictionaries and sets are containers for sequential data.
  - All members of a set have to be hashable, just like dictionary keys.
  - Integers, floating point numbers, tuples, and strings are hashable; dictionaries, lists, and other sets (except frozensets) are not.

Table
. Built-in objects preview

Object type	-	Example literals/creation
Numbers	-	1234, 3.1415, 3+4j, Decimal, Fraction
Strings	-	'spam', "guido's", b'a\x01c'
Lists	-	[1, [2, 'three'], 4]
Dictionaries	-	{'food': 'spam', 'taste': 'yum'}
Tuples	-	(1, 'spam', 4, 'U')
Files	-	myfile = open('eggs', 'r')
Sets	-	set('abc'), {'a', 'b', 'c'}
Other core types	-	Booleans, types, None
Program unit types	-	Functions, modules, classes
Implementation-related types	-	Compiled code, stack tracebacks

Built-in Data types

- Python's built-in (or standard) data types can be grouped into several classes. - **numeric types, sequences, sets and mappings**
  - boolean: the type of the built-in values True and False. Useful in conditional expressions, and anywhere else you want to represent the truth or falsity of some condition. Mostly interchangeable with the integers 1 and 0. In fact, conditional expressions will accept values of any type, treating special ones like boolean False, integer 0 and the empty string "" as equivalent to False, and all other values as equivalent to True. But for safety's sake, it is best to only use boolean values in these places.
- Numeric types:
  - int: Integers; equivalent to C longs in Python 2.x, non-limited length in Python 3.x
  - long: Long integers of non-limited length; exists only in Python 2.x
  - float: Floating-Point numbers, equivalent to C doubles
  - complex: Complex Numbers

Built-in Data types

- Sequences:
  - **str:** String; represented as a sequence of 8-bit characters in Python 2.x, but as a sequence of Unicode characters (in the range of U+0000 - U+10FFFF) in Python 3.x
  - **bytes:** a sequence of integers in the range of 0-255; only available in Python 3.x
  - **byte array:** like bytes, but mutable (see below); only available in Python 3.x
  - **list**
  - tuple
- Sets:
  - set: an unordered collection of unique objects; available as a standard type since Python 2.6
  - frozen set: like set, but immutable (see below); available as a standard type since Python 2.6
- Mappings:
  - dict: Python dictionaries, also called hashmaps or associative arrays, which means that an element of the list is associated with a definition, rather like a Map in Java

Strings

- They can be enclosed in single quotes ('...') or double quotes ("...") with the same result.

```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

- If you don't want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

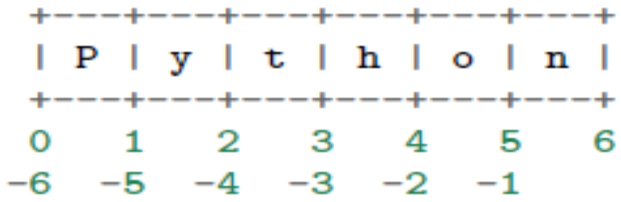
Strings and formatting

- Python string is a contiguous sequence of Unicode characters. Strings - characters are strings of length 1

```
i = 10

d = 3.1415926

s = "I am a string!"

print "%d\t%f\t%s" % (i, d, s)

print "newline\n"

print "no newline"
```

- In order to convert a character value to ASCII code, the ord() function is used, and for converting ASCII code to character, the chr() function is used

Strings



- "hello"+"world" "helloworld" # concatenation
- 'Py' 'thon' 'Python'
- "hello"*3 "hellohellohello“ # repetition
- "hello"[0] "h" # indexing
- "hello"[-1] "o" # (from end)
- "hello"[1:4] "ell" # slicing
- len("hello") 5 # size
- "hello" < "jello" 1 # comparison
- "e" in "hello" 1 # search
- New line: "escapes: \n "
- Line continuation: triple quotes ”’
- Quotes: ‘single quotes’, "raw strings"

Methods in string

- upper()
- lower()
- capitalize()
- count(s)
- find(s)
- rfind(s)
- index(s)

- strip(), lstrip(), rstrip()
- replace(a, b)
- expandtabs()
- split()
- join()
- center(), ljust(), rjust()
- If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

- Python strings cannot be changed — they are *immutable*. Therefore, assigning to an indexed position in the string results in an error.

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

Tuples

- What is a tuple?
  - A tuple is an ordered collection which cannot be modified once it has been created - immutable. Index starts from 0.
  - In other words, it's a special array, a read-only array.
- How to make a tuple? In round brackets
- A tuple consists of a number of values separated by commas.
  - E.g.,

```
>>> t = ()
>>> t = (1, 2, 2, 3)
>>> t = (1, )
>>> t = 1,
>>> a = (1, 2, 3, 4, 5)
>>> print a[1]  # 2
```

Operations in Tuple

- Indexing e.g., T[i]
- Slicing e.g., T[1:5]
- Concatenation e.g., T + T
- Repetition e.g., T * 5
- Membership test e.g., 'a' in T
- Length e.g., len(T)
- any e.g. True if any element is True
- all e.g. True if all elements are True
- Tuple methods -
  - tpl.count(23)
  - tpl.index(22)
- Unpacking y = (10, 20, *t, 30) = (10, 20, 12345, 54321, 'hello!', 30)

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Tuples and sequences

- lists, strings, **tuples**: examples of *sequence* type
- tuple = values separated by commas

```
>>> t = 123, 543, 'bar'
>>> t[0]
123
>>> t
(123, 543, 'bar')
```

- Tuples are *immutable*, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of namedtuples).
- `S = ([1, 2, 3, 4], [4, 5], 'Ocelot')`
- `S[1][1]=45`
- `p=s[1]`
- `p[1]=100`

Tuples

- Tuples may be nested

```
>>> u = t, (1,2)
>>> u
((123, 542, 'bar'), (1,2))
```

- kind of like structs, but no element names:
  - (x,y) coordinates
  - database records
- like strings, immutable → can't assign to individual items

Tuples

- Empty tuples: ()

```
>>> empty = ()
>>> len(empty)
0
```

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

- one item → trailing comma

```
>>> singleton = 'foo',
```

- Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

Tuples

- sequence unpacking → distribute elements across variables

```
>>> t = 123, 543, 'bar'
>>> x, y, z = t
>>> x
123
```

- The statement t = 12345, 54321, 'hello!'
  - is an example of *tuple packing*: the values 12345, 54321 and 'hello!' are packed together in a tuple.
- packing always creates tuple
- unpacking works for any sequence on the right hand side

Container Data Type: List

- List:
  - A container that holds a number of other objects, in a given order
  - Defined in square brackets, index starts from 0
  - Flexible arrays

```
a = [1, 2, 3, 3, 4, 5, 6.5, "Seven"]
Print(a)
print a[1]  # number 2
some_list = []
some_list.append("foo")
some_list.append(12)
print len(some_list)  # 2
```

Lists

- Lists are *mutable*, and their elements are usually homogeneous and are accessed by iterating over the list.
- lists can be heterogeneous
  - `a = [‘computer', ‘mobile', 100, 1234, 2*2]`
- Lists can be indexed and sliced:
  - `a[0]` → computer
  - `a[:2]` → [‘computer', ‘mobile']
- Lists can be manipulated
  - `a[2] = a[2] + 23`
  - `a[0:2] = [1,12]`
  - `a[0:0] = []`
  - `len(a)` → 5

```
for b in a :
    print(b)
```

More list operations

```
>>> a = range(5)                # [0,1,2,3,4]
>>> a.append(5)                 # [0,1,2,3,4,5]
>>> a.pop()                     # [0,1,2,3,4]
5
>>> a.insert(0, 5.5)            # [5.5,0,1,2,3,4]
>>> a.pop(0)                    # [0,1,2,3,4]
5.5
>>> a.reverse()                       # [4,3,2,1,0]
>>> a.sort()                    # [0,1,2,3,4]
>>>a.sort(reverse=True), print(lst(::-1))
>>> a=a+b
>>>l=list('Africa')
```

@ Shubin Liu

Nested List

- List in a list
- E.g.,
  - >>> s = [1,2,3]
  - >>> t = [‘begin’, s, ‘end’]
  - >>> t
  - [‘begin’, [1, 2, 3], ‘end’]
  - >>> t[1][1]
  - 2
  - >>>res= ‘a’ in lst
  - >>>res= ‘a’ not in lst
  - >>>print(lst1 is lst2)
  - >>>print(s<t)

- Shallow copy or Aliasing
  - lst2=lst1
  - lst2[0]=100
  - print(lst2[0],lst1[0])
    - – 100 100
- Deep copy or Cloning
  - lst2=[ ]
  - lst2=lst2+lst1
  - lst1[0]=100
  - print(lst2[0],lst1[0])
    - – 10 100
  - lst=[]
  - If not lst:
    - print(“Empty list’)

List methods

- `append(x)` – Equivalent to a[len(a):] = [x].
- `extend(L)`
  - append all items in list (like Tcl lappend)
  - Equivalent to a[len(a):] = iterable.
- `insert(i,x)`- a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
- `remove(x)` – It raises a ValueError if there is no such item.
- `pop([i]), pop()`
  - Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.
  - create stack (FIFO), or queue (LIFO) → pop(0)
- `index(x`[, *start*[, *end*]])
  - return the index for value *x*
  - Return zero-based index in the list of the first item whose value is equal to *x*. Raises a ValueError if there is no such item.

List methods

- `count(x)`
  - how many times x appears in list
- `sort(), sort(key=None, reverse=False)`
  - sort items in place
- sorted()
  - Return another sorted list, list remains unchanged
- `reverse()`
  - reverse list in place
- `clear()` – Equivalent to del a[:].
- `copy()`
  - Return a shallow copy of the list. Equivalent to a[:].
- del()
  - del(lst[2:5]), del(a[:])
  - lst3=lst2=lst1
  - lst1=[ ], lst2[:]=[ ]

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)   # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

- Methods like insert, remove or sort that only modify the list have no return value printed – they return the default None. This is a design principle for all mutable data structures in Python.

List as Stack

- The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index.

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

List as Queue

- It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).
- To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends.

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Comprehensions

- https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html
- https://realpython.com/list-comprehension-python/
- Looping or iterating over items and assigning them to a container like list, set or dictionary.
- This container can not be tuple.
- An expression followed by for clause, and zero or more for or if clauses.
- Nested comprehension
  - lst = [a+b for a in [1,2,3] for b in [3, 4, 5] = [4, 5, 6, 5, 6, 7, 6, 7, 8]
  - lst = [[a+b for a in [1,2,3]] for b in [3,4,5]] = [[4,5,6], [5,6,7], [6,7,8]]

Collections

- https://docs.python.org/3/library/collections.html

List comprehensions (2.0)

- List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
squares = list(map(lambda x: x**2, range(10)))
```

- Create lists without `map(), filter(), lambda`

```
squares = [x**2 for x in range(10)]
```

- = expression followed by for clause + zero or more for or if clauses

```
>>> vec = [2,4,6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec}
[{2: 4}, {4: 16}, {6: 36}]
```

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

List comprehensions

- cross products:

```
>>> vec1 = [2,4,6]
>>> vec2 = [4,3,-9]
>>> [x*y for x in vec1 for y in vec2]
[8,6,-18, 16,12,-36, 24,18,-54]
>>> [x+y for x in vec1 and y in vec2]
[6,5,-7,8,7,-5,10,9,-3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8,12,-54]
```

- A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.
- The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.
- The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

List comprehensions

- can also use `if`:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

If the expression is a tuple (e.g. the (x, y) in the previous example), it must be parenthesized.

```
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

del – removing list items

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)

```
>>> a = [-1,1,66.6,333,333,1234.5]
>>> del a[0]
>>> a
[1,66.6,333,333,1234.5]
>>> del a[2:4]
>>> a
[1,66.6,1234.5]
>>> del a[:]
>>> a
[]
>>> del a
```

List vs. Tuple

- What are common characteristics?
  - Both store arbitrary data objects
  - Both are of sequence data type
- What are differences?
  - Tuple **doesn't allow modification**
  - Tuple doesn't have methods
  - Tuple supports format strings
  - Tuple supports variable length parameter in function call.
  - Tuples **slightly faster**

Set

- An unordered collection of unique and mutable objects that supports operations corresponding to mathematical set theory.
- No slice, no index. No merging+, No embedding

```
>>> x = set('abcde')
>>> y = set('bdxyz')
>>> x
set(['a', 'c', 'b', 'e', 'd'])
```

Iterable containers

```
>>> for item in set('abc'): print(item * 3)
aaa
ccc
Bbb
>>>print(*x) -- unpacking
```

- with no duplicate elements
- an empty set = set()

```
>>> 'e' in x # Membership
True
>>> x – y # Difference
set(['a', 'c', 'e'])
>>> x | y # Union
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])
>>> x & y # Intersection
set(['b', 'd'])
>>> x ^ y # Symmetric difference (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])
>>> x > y, x < y # Superset, subset
(False, False)
>>>u=s.copy()  -- deep copy (cloning)
```

- issuperset, >= issubset , <=--

Sets

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
```

- >>> **engineers = {'bob', 'sue', 'ann', 'vic'}**
- >>>engineers.add(‘John’)
- >>> **managers = {'tom', 'sue'}**
- >>> **engineers – managers** *# Engineers who are not managers*
- {'vic', 'bob', 'ann'}
- >>> **{'bob', 'sue'} < engineers** *# Are both engineers? (subset)*
- True
- >>> **managers ^ engineers** *# Who is in one but not both?*
- {'vic', 'bob', 'ann', 'tom'}
- set comprehensions are also supported

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

- While storing an element, its hash value is computed using a hashing method to determine where it should be stored.

Immutable constraints and frozen sets and Comprehensions

```
>>> S
{1.23}
>>> S.add((1, 2, 3))
>>> S # No list or dict, but tuple okay, Only mutable objects work in a set
{1.23, (1, 2, 3)}
>>> (1, 2, 3) in S # Membership: by complete values
True
```

Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the *frozenset* built-in call works just like set but creates an immutable set that cannot change and thus can be embedded in other sets.

Set comprehensions run a loop and collect the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression.

```
>>> {x ** 2 for x in [1, 2, 3, 4]}
{16, 1, 4, 9}
```

Sets can be used to filter duplicates out of other collections.

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)   {1, 2, 3, 4, 5}
>>> L = list(set(L)) # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
```

Frozenset

- The relationship between frozenset and set is like the relationship between tuple and list.
- Frozenset is an immutable version of set. A frozenset is basically the same as a set, except that it is immutable - once it is created, its members cannot be changed.
- Since they are immutable, they are also hashable, which means that frozensets can be used as members in other sets and as dictionary keys. frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop, etc.) are available.

```
>>> frozen=frozenset(['life','universe','everything'])
>>> fs = frozenset([2, 3, 4])
>>> s1 = set([fs, 4, 5, 6])
>>> s1
set([4, frozenset([2, 3, 4]), 6, 5])
>>> fs.intersection(s1)
frozenset([4])
>>> fs.add(6)
Traceback (most recent call last): File "<stdin>", line 1, in <module> AttributeError:
'frozenset' object has no attribute 'add'
```

Other data types

- Python also has other types of sequences, though these are used less frequently and need to be imported from the standard library before being used.
  - **Array** - A typed-list, an array may only contain homogeneous values.
  - **collections.defaultdict** - A dictionary that, when an element is not found, returns a default value instead of error.
  - **collections.deque** - A double ended queue, allows fast manipulation on both sides of the queue.
  - **Heapq** - A priority queue.
  - **Queue** - A thread-safe multi-producer, multi-consumer queue for use with multi-threaded programs. Note that a list can also be used as queue in a single-threaded
- **3rd party data structure**[edit]
- Some useful data types in Python do not come in the standard library. Some of these are very specialized in their use.
  - **numpy.array** - useful for heavy number crunching Sorteddict - like the name says, a sorted dictionary

Dictionaries

- Dictionaries: curly brackets, mutable
  - What is dictionary?
    - Refer value through key; "**associative arrays**"
  - Like an array indexed by a string
  - Keys can not be changed in place
  - An unordered set of *key: value* pairs
  - *Values* of any type; keys of almost any type
    - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

```
d = { "foo" : 1, "bar" : 2 }
print d["bar"]    # 2
some_dict = {}
some_dict["foo"] = "yow!"
print some_dict.keys() # ["foo"]
```

- **Concatenation, merging, comparison – do not work**

Dictionary details

- Keys must be **immutable, unique**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing
  - For repeated pairs,, only one pair is stored.
  - Dictionary preserved insertion order.

Dictionaries

- like Tcl or awk associative arrays
- indexed by keys
- keys are any immutable type: e.g., tuples, strings, numbers
- but not lists (mutable!)
- uses 'key: value' notation, can be nested

```
>>> tel = {'hgs' : 7042, 'lennox': 7018}
>>> tel['cs'] = 7000
>>> tel
```

- `C.get`
- `C.update(d)`
- `C.popitem – LIFO order`
- `C.clear()`

Dictionaries

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

- use has_key() to check for existence

```
>>> tel.has_key('foo')
0
```

- It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.
- Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the in keyword.

Dictionaries

- Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys.
- Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.
- You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().
- It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary).
- A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary.

Methods in Dictionary

- keys()
- values()
- items()
- has_key(key)
- clear()
- copy()
- get(key[,x])
- setdefault(key[,x])
- update(D)
- popitem()
- for k in courses.keys(): -- courses,values()
  - print(k)
- for k,v in courses.items:
  - print(k,v)

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Constructors and comprehensions

- The dict() constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

- dict comprehensions can be used to create dictionaries from arbitrary key and value expressions

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

- When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

- Unpacking –
  - Combined = {**animals, **birds}
  - For * , only keys are unpacked
- Different keys, with same values –
  - lst = [12, 13, 14, 15]
  - d=dict.fromkeys(lst, 25)

Make a dictionary that maps keys to more than one values

- You need to store the multiple values in another container such as a list or set.

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}

e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

- Use a list if you want to preserve the insertion order of the items. Use a set if you want to eliminate duplicates (and don't care about the order).
- To easily construct such dictionaries, you can use defaultdict in the collections module. A feature of defaultdict is that it automatically initializes the first value so you can simply focus on adding items.

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...
```

One caution with defaultdict is that it will automatically create dictionary entries for keys accessed later on (even if they aren't currently found in the dictionary).

Ordering a dictionary

- To control the order of items in a dictionary, you can use an OrderedDict from the collections module. It exactly preserves the original insertion order of data when iterating.

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

- An OrderedDict internally maintains a doubly linked list that orders the keys according to insertion order.
- Be aware that the size of an OrderedDict is more than twice as large as a normal dictionary due to the extra linked list that's created.

Comparing sequences

- unlike C, can compare sequences (lists, tuples, ...)
- lexicographical comparison:
  - compare first; if different → outcome
  - continue recursively
  - subsequences are smaller
  - strings use ASCII comparison
  - can compare objects of different type, but by type name (list < string < tuple)

Comparing sequences

(1,2,3) < (1,2,4)

[1,2,3] < [1,2,4]

'ABC' < 'C' < 'Pascal' < 'Python'

(1,2,3) == (1.0,2.0,3.0)

(1,2) < (1,2,-1)

Comparing Sequences and Other Types

- Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted.
- Lexicographical ordering for strings uses the Unicode code point number to order individual characters.

```
(1, 2, 3)                < (1, 2, 4)
[1, 2, 3]                < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)             < (1, 2, 4)
(1, 2)                   < (1, 2, -1)
(1, 2, 3)               == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))     < (1, 2, ('abc', 'a'), 4)
```

Data Type Wrap Up

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

Data Type Wrap Up

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Lambda forms – Functional Programming

- anonymous functions
- may not work in older versions

```
def make_incrementor(n):
   return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

Functional programming tools

- filter(*function, sequence*)
  - filter does the work of a list-comprehension if

```
def f(x): return x%2 != 0 and x%3 == 0
filter(f, range(2,25))
```

```
def is_even(x):
    """True if x is even, False if x is odd"""
    return x % 2 == 0
```

```
x_evens = [x for x in xs if is_even(x)]
x_evens = filter(is_even, xs)
list_evener = partial(filter, is_even)
x_evens = list_evener(xs)
```

```
# [2, 4]
# same as above
# *function* that filters a list
# again [2, 4]
```

- map(*function, sequence*)
  - call function for each item
  - return list of return values

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

- reduce(*function, sequence*)
  - return a single value
  - call binary function on the first two items
  - then that result with the third and so on
  - iterate

```
x_product = reduce(multiply, xs)
list_product = partial(reduce, multiply)
x_product = list_product(xs)
```

```
# = 1 * 2 * 3 * 4 = 24
# *function* that reduces a list
# again = 24
```

Language comparison

		Tcl	Perl	Python	JavaScript	Visual Basic
Speed	development	✓	✓	✓	✓	✓
	regexp	✓	✓	✓		
breadth	extensible	✓		✓		✓
	embeddable	✓		✓		
	easy GUI	✓		✓ (Tk)		✓
	net/web	✓	✓	✓	✓	✓
enterprise	cross-platform	✓	✓	✓	✓	
	I18N	✓		✓	✓	✓
	thread-safe	✓		✓		✓
	database access	✓	✓	✓	✓	✓

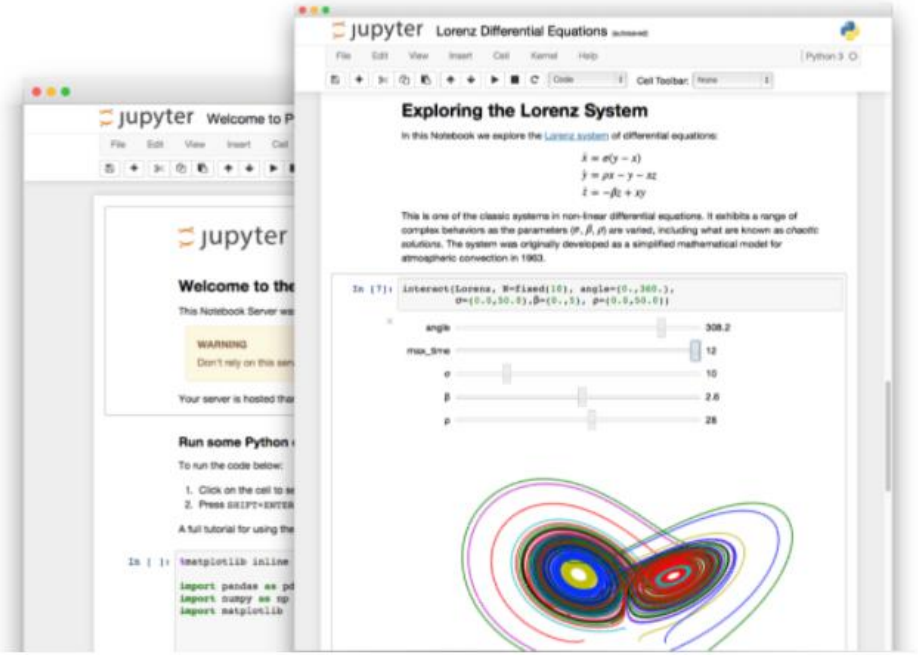
Python: Pros & Cons

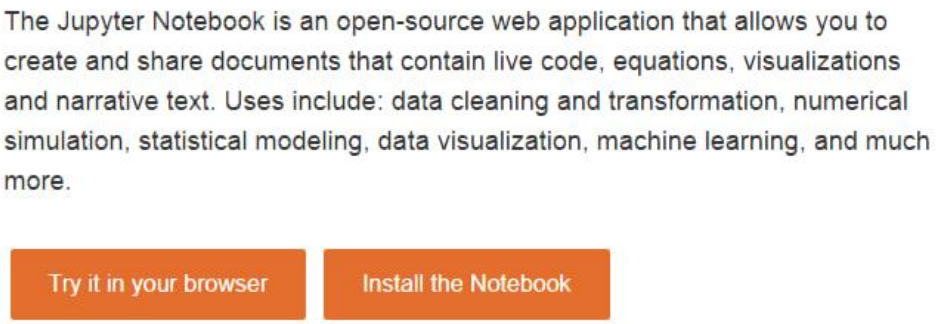
- Pros
  - Free availability (like Perl, Python is open source).
  - Stability (Python is in release 2.6 at this point and, as I noted earlier, is older than Java).
  - Very easy to learn and use
  - Good support for objects, modules, and other reusability mechanisms.
  - Easy integration with and extensibility using C and Java.
- Cons
  - Smaller pool of Python developers compared to other languages, such as Java
  - Lack of true multiprocessor support
  - Absence of a commercial support point, even for an Open Source project (though this situation is changing)
  - Software performance slow, not suitable for high performance applications

DATA SCIENCE APPLICATIONS

Jupyter Notebook

- *"The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more."*
- *–description from [Project Jupyter](#)*





What is PIP install in Python?

- PIP is a package management system used to install and manage software packages written in Python.
- It stands for “preferred installer program” or “Pip Installs Packages.”
- PIP for Python is a utility to manage PyPI package installations from the command line.
- As of January 2020, the Python Package Index (PyPI), the official repository for third-party Python software, contains over 287,000 packages with a wide range of functionality.

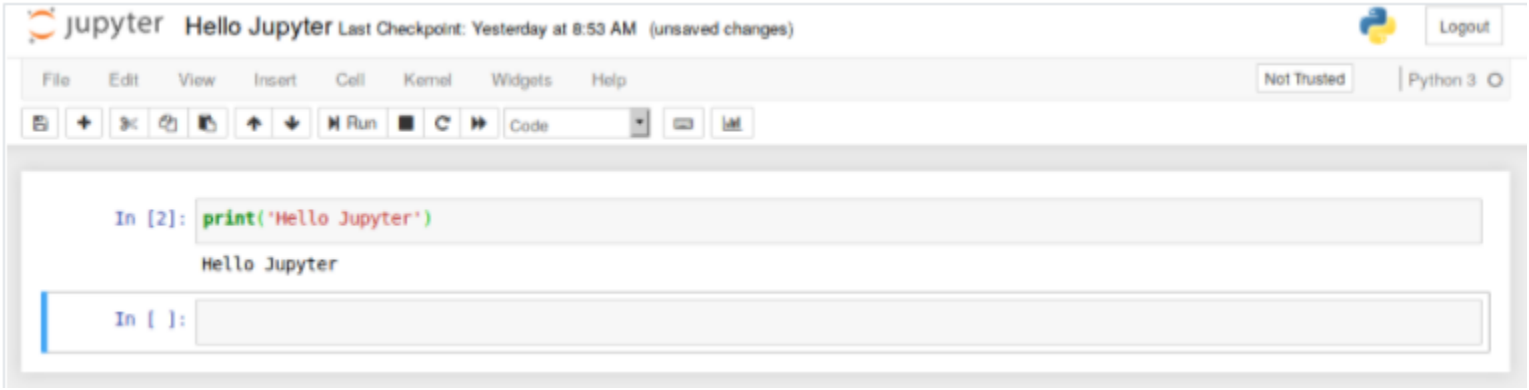
Shell

```
$ pip install jupyter
```

Shell

```
$ jupyter notebook
```

URL: http://localhost:8888/tree



Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance
- many other python libraries are built on NumPy

Link:

http://www.numpy.org/

Python Libraries for Data Science

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack
- built on NumPy

Link:
https://www.scipy.org/scipylib/

Python Libraries for Data Science

Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.

- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

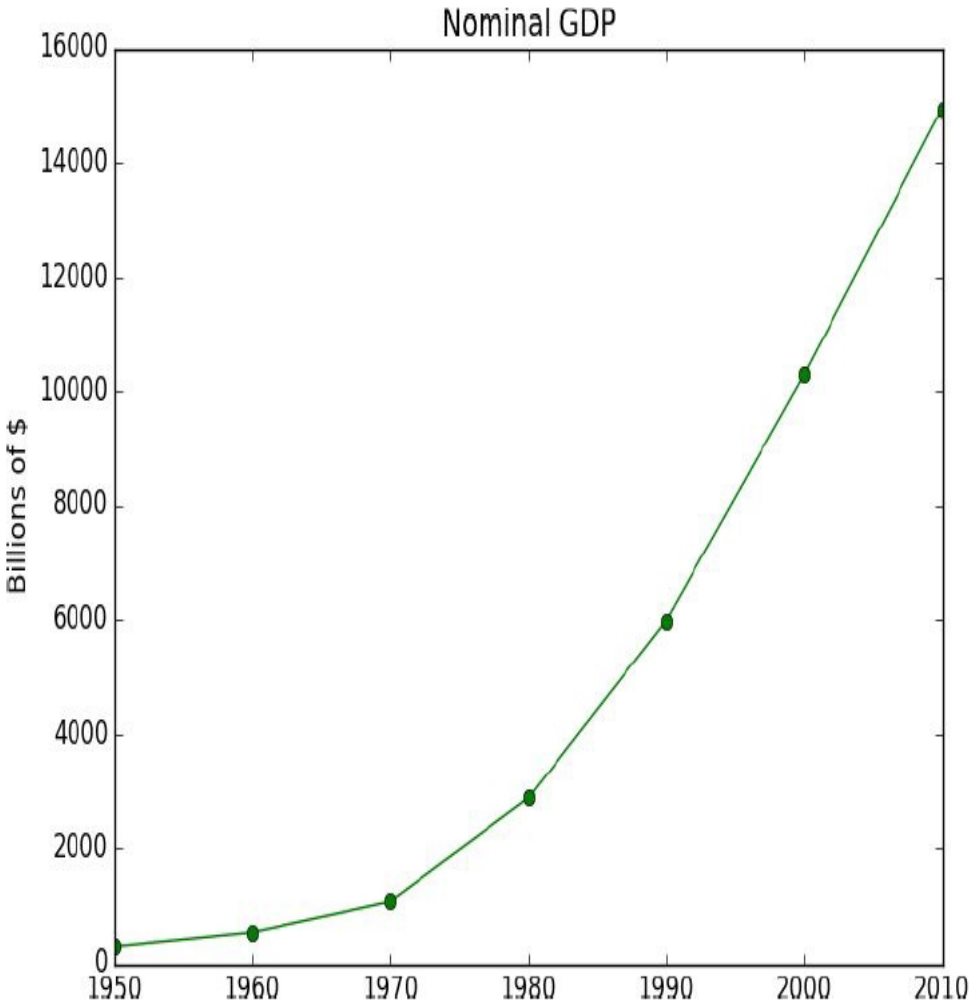
Visualizing Data

- There are two primary uses for data visualization:
- To *explore* data
- To *communicate* data

matplotlib

- matplotlib.pyplot module

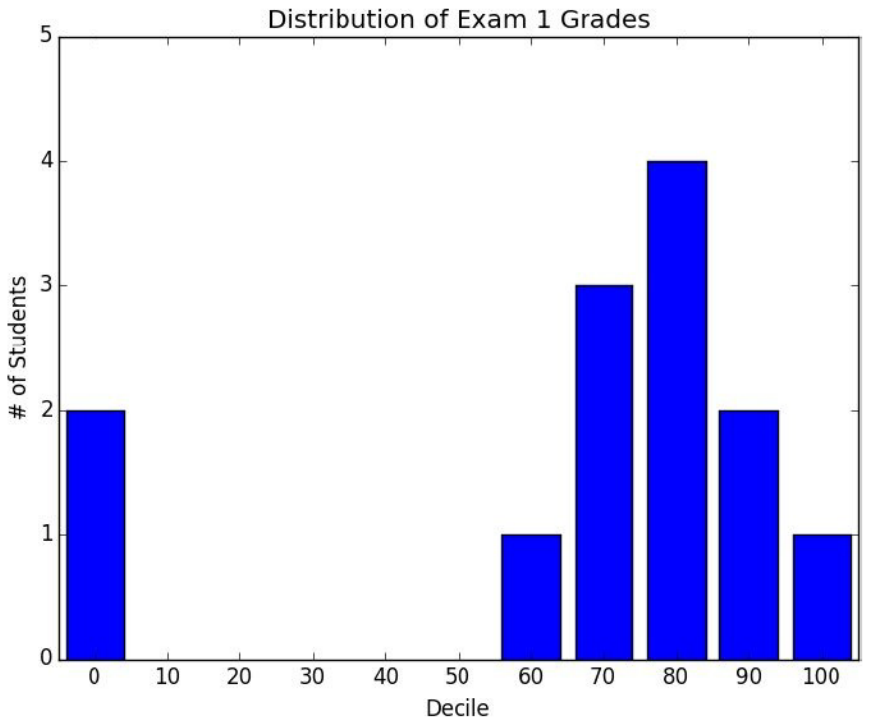
```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid'

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```



Using a bar chart for a histogram

- A bar chart is a good choice when you want to show how some quantity varies among some discrete set of items.
- A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are distributed.

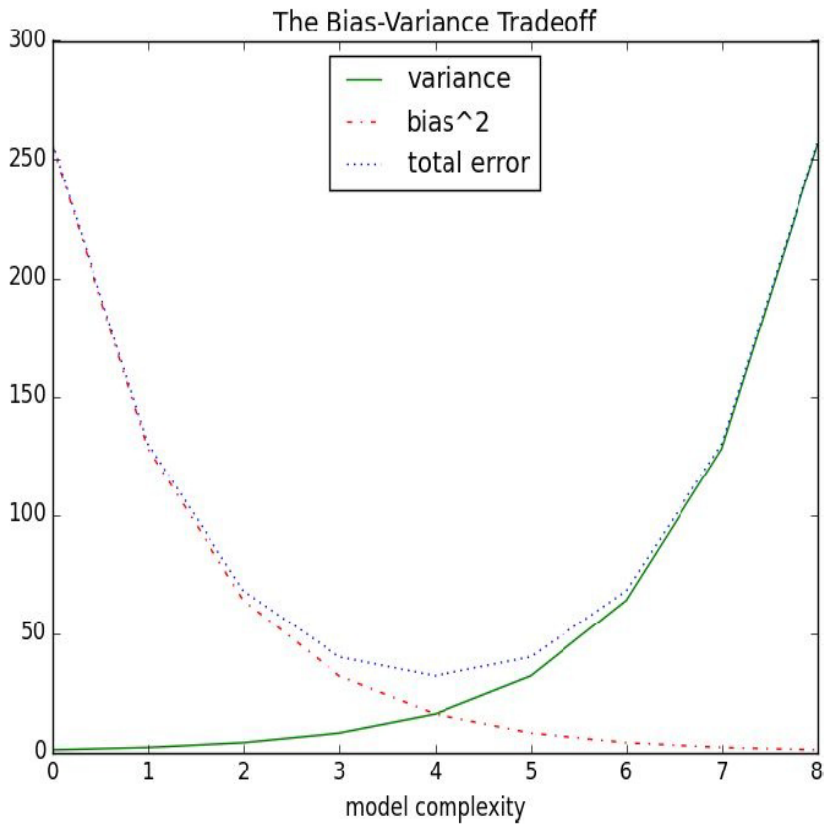
```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # shift each bar to the left by 4
        histogram.values(),               # give each bar its correct height
        8)                                # give each bar a width of 8

plt.axis([-5, 105, 0, 5])                 # x-axis from -5 to 105,
                                          # y-axis from 0 to 5

plt.xticks([10 * i for i in range(11)])   # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```



Line Charts

- As we saw already, we can make line charts using plt.plot(). These are a good choice for showing trends.

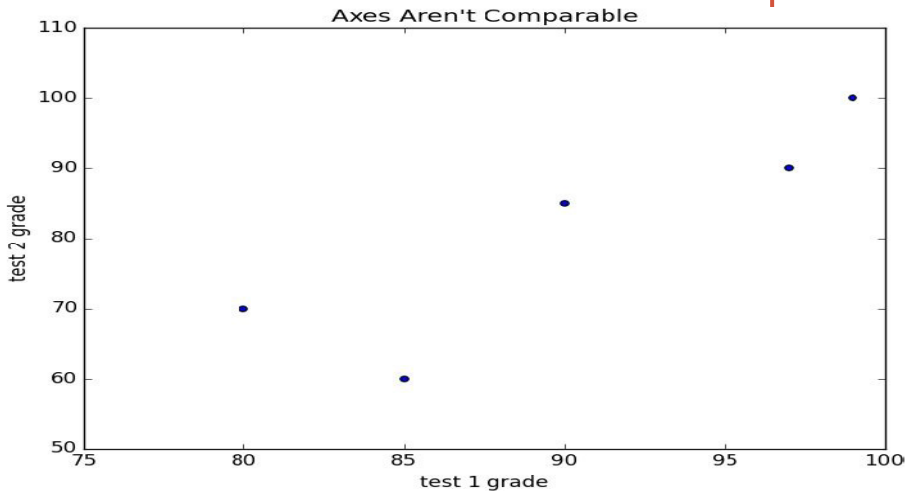
```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# we can make multiple calls to plt.plot
# to show multiple series on the same chart
plt.plot(xs, variance,     'g-',  label='variance')    # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # red dot-dashed line
plt.plot(xs, total_error,  'b:',  label='total error') # blue dotted line

# because we've assigned labels to each series
# we can get a legend for free
# loc=9 means "top center"
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.title("The Bias-Variance Tradeoff")
plt.show()
```



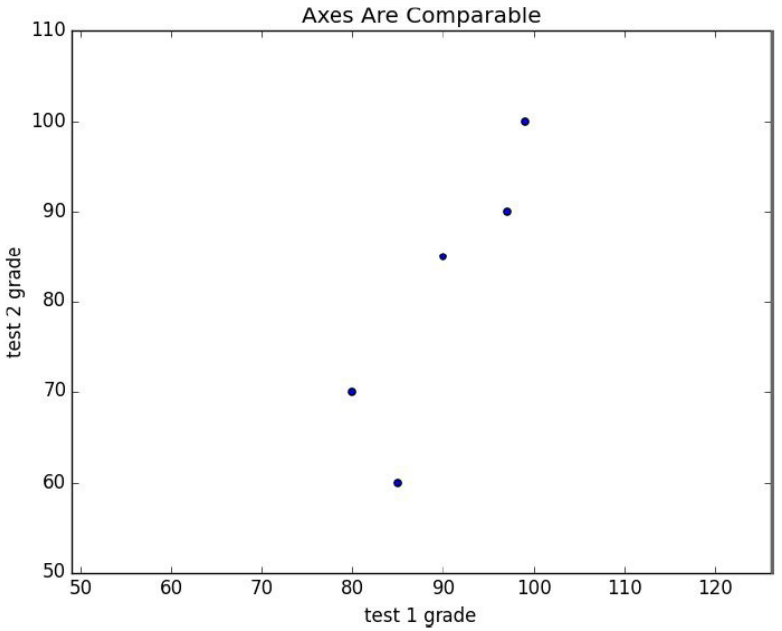
Scatterplots

- A scatterplot is the right choice for visualizing the relationship between two paired sets of data.
- If you're scattering comparable variables, you might get a misleading picture if you let matplotlib choose the scale.

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```



- If we include a call to plt.axis("equal"), the plot more accurately shows that most of the variation occurs on test 2.



More tools

- **seaborn** is built on top of **matplotlib** and allows to easily produce prettier (and more complex) visualizations.
- **D3.js** is a JavaScript library for producing sophisticated interactive visualizations for the web.
- **Bokeh** is a newer library that brings D3-style visualizations into Python.
- **ggplot** is a Python port of the popular R library **ggplot2**, which is widely used for creating “publication quality” charts and graphics.

What is pandas ?

- ***Pandas*** is Python package for **data analysis**.
- It Provides built-in data structures which simplify the manipulation and analysis of data sets.
- Pandas is easy to use and powerful, but “with great power comes great responsibility”
- http://pandas.pydata.org/pandas-docs/stable/
- Pandas Basics
  - Series
  - DataFrame
  - Creating a DataFrame from a dict
  - Get a column, Get rows

Pandas: Essential Concepts

- A **Series** is a named Python list (dict with list as value).

```
{ 'grades' : [50,90,100,45] }
```

- A **DataFrame** is a collection of Series (dict of series):

```
{  { 'names' : ['bob','ken','art','joe']}
    { 'grades' : [50,90,100,45] }
  }
```

- Reading Data with Pandas
  - Pandas can read a variety of formats!

Loading Python Libraries

In [ ]:
```
#Import Python Libraries
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mpl
import seaborn as sns
```

Reading data using pandas

In [ ]:
```
#Read csv file
df =
pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/
Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1',
index_col=None, na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
```

Exploring data frames

In [3]:
```
#List first 5 records
df.head()
```

Out[

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

Data Frame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Data Frames methods

Unlike attributes, python methods have *parenthesis*.
All attributes and methods can be listed with a *dir()* function:
`dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Data Frames *groupby* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group
- Similar to dplyr() function in R

In [ ]:
```
#Group data using rank
df_rank = df.groupby(['rank'])
```

In [ ]:
```
#Calculate mean value for each numeric column per each group
df_rank.mean()
```

rank	phd	service	salary
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than $120K:

In [ ]:
```
#Calculate mean salary for each professor rank:
df_sub = df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;

< less; <= less or equal;

== equal; != not equal;

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

In [ ]:
```
#Select column salary:
df['salary']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

In [ ]:
```
#Select column salary:
df[['rank','salary']]
```

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

In [ ]:
```
#Select rows by their labels:
df_sub.loc[10:20,['rank','sex','salary']]
```

Out[ ]:

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

In [ ]:
```
#Select rows by their labels:
df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[ ]:

	rank	service	sex	salary
26	Prof	19	Male	148750
27	Prof	43	Male	155865
29	Prof	20	Male	123683
31	Prof	21	Male	155750
35	Prof	23	Male	126933
36	Prof	45	Male	146856
39	Prof	18	Female	129000
40	Prof	36	Female	137000
44	Prof	19	Female	151768
45	Prof	25	Female	140096

Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

In [ ]:
```
# Create a new data frame from the original sorted by the column Salary
df_sorted = df.sort_values( by ='service')
df_sorted.head()
```

Out[ ]:

	rank	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

Missing Values

Missing values are marked as NaN

In [ ]:
```
# Read a dataset with missing values
flights =
pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/flights.csv")
```

In [ ]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[ ]:

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWR	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWR	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

```
In [ ]: flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

Out[ ]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

What is Numpy?

- Numpy, Scipy, and Matplotlib provide MATLAB-like functionality in python.
- Numpy Features:
  - Typed multidimentional arrays (matrices)
  - Fast numerical computations (matrix math)
  - High-level math functions
- Python does numerical computations slowly.
- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - Numpy takes ~0.03 seconds

NumPy Overview

1. Arrays
2. Shaping and transposition
3. Mathematical Operations
4. Indexing and slicing
5. Broadcasting

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- **Vectors**
- **Matrices**
- Images
- Tensors
- ConvNets

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Arrays

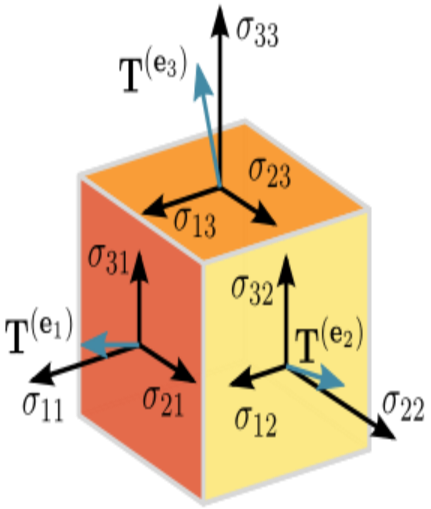
Structured lists of numbers.

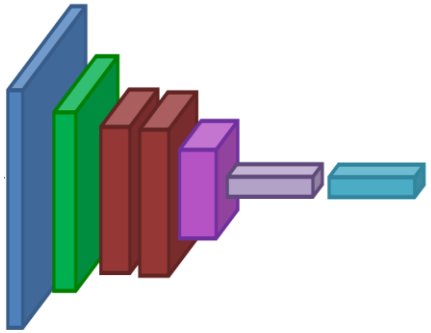
- Vectors
- Matrices
- **Images**
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- **Tensors**
- **ConvNets**





Arrays, Basic Properties

```
import numpy as np
a                                                    =
np.array([[1,2,3],[4,5,6]],dtype=np.float3
2)
print a.ndim, a.shape, a.dtype
```

1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: np.uint8, np.int64, np.float32, np.float64
3. Arrays are dense. Each element of the array exists and has the same type.

Arrays, creation

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

Saving and loading arrays

```
np.savez('data.npz', a=a)
data = np.load('data.npz')
a = data['a']
```

1. NPZ files can hold multiple arrays
2. np.savez_compressed similar.

Saving and Loading Images

SciPy: `skimage.io.imread`, `skimage.io.imsave`

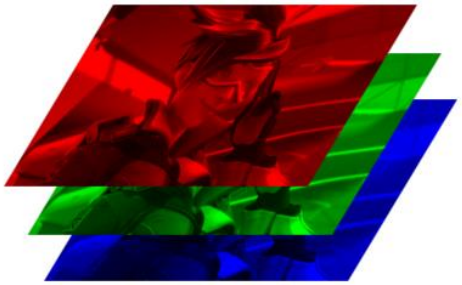
height x width x RGB

PIL / Pillow: `PIL.Image.open`, `Image.save`

width x height x RGB

OpenCV: `cv2.imread`, `cv2.imwrite`

height x width x BGR



Mathematical operators

- **Arithmetic operations are element-wise**
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Math, upcasting

Just as in Python and Java, the result of a math operator is cast to the more general or precise datatype.

uint64 + uint16 => uint64

float32 / int32 => float32

Warning: upcasting does not prevent overflow/underflow. You must manually cast first.

Use case: images often stored as uint8. You should convert to float32 or float64 before doing math.

Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
>>> a
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> np.sqrt(a)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

Indexing

```
x[0,0]    # top-left element
x[0,-1]   # first row, last column
x[0,:]    # first row (many entries)
x[:,0]    # first column (many entries)
```

Notes:

- Zero-indexing
- Multi-dimensional indices are comma-separated (i.e., a tuple)

Python Slicing

Syntax: start:stop:step

```
a = list(range(10))
a[:3] # indices 0, 1, 2
a[-3:] # indices 7, 8, 9
a[3:8:2] # indices 3, 5, 7
a[4:1:-1] # indices 4, 3, 2 (this one
is tricky)
```

Axes

```
a.sum() # sum all entries
a.sum(axis=0) # sum over rows
a.sum(axis=1) # sum over columns
a.sum(axis=1, keepdims=True)
```

1. Use the axis parameter to control which axis NumPy operates on
2. Typically, the axis specified will disappear, keepdims keeps all dimensions

Broadcasting

```
a = a + 1 # add one to every element
```

When operating on multiple arrays, broadcasting rules are used.

Each dimension must match, from right-to-left

1. Dimensions of size 1 will broadcast (as if the value was repeated).
2. Otherwise, the dimension must have the same shape.
3. Extra dimensions of size 1 are added to the left as needed.

WORKING WITH DATA

Getting Data

- **Finding APIs**
- If you need data from a specific site, look for a developers or API section of the site for details, and try searching the Web for "*python __ api*" to find a library.
- Twitter is a fantastic source of data to work with. And you can get access to its data through its API. To interact with the Twitter APIs one can use the *Twython library (pip install twython).*

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

```
haithemnyc: Data scientists with the technical savvy &amp;amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

Exploring Data

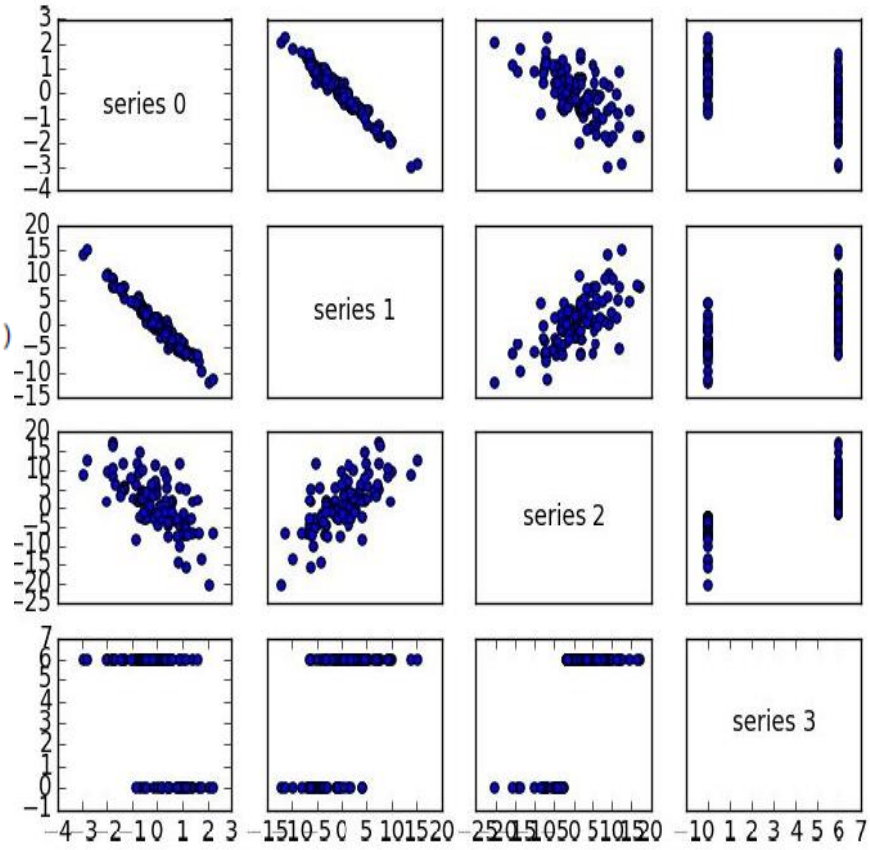
- With many dimensions, you'd like to know how all the dimensions relate to one another. A simple approach is to look at the correlation matrix, in which the entry in row i and column j is the correlation between the ith dimension and the jth dimension of the data.

```
def correlation_matrix(data):
    """returns the num_columns x num_columns matrix whose (i, j)th entry
    is the correlation between columns i and j of data"""

    _, num_columns = shape(data)

    def matrix_entry(i, j):
        return correlation(get_column(data, i), get_column(data, j))

    return make_matrix(num_columns, num_columns, matrix_entry)
```

Scatterplot Matrix

- A more visual approach (if you don't have too many dimensions) is to make a scatterplot matrix showing all the pairwise scatterplots. To do that one needs to use **plt.subplots(),** which allows to create subplots of the chart.

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # scatter column_j on the x-axis vs column_i on the y-axis
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

        # unless i == j, in which case show the series name
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # then hide axis labels except left and bottom charts
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# fix the bottom right and top left axis labels, which are wrong because
# their charts only have text in them
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```



Cleaning and Munging

- Real-world data is dirty. Often you'll have to do some work on it before you can use it.
- We have to convert strings to floats or ints before we can use them.

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

```
import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)

for row in data:
    if any(x is None for x in row):
        print row
```

Rescaling

- Many techniques are sensitive to the *scale* of your data. For example, imagine that you have a data set consisting of the heights and weights of hundreds of data scientists, and that you are trying to identify *clusters* of body sizes.
- To start with, we'll need to compute the mean and the standard_deviation for each column and then use them to create a new data matrix.

```
def scale(data_matrix):
    """returns the means and standard deviations of each column"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
              for j in range(num_cols)]
    return means, stdevs
```

```
def rescale(data_matrix):
    """rescales the input data so that each column
    has mean 0 and standard deviation 1
    leaves alone columns with no deviation"""
    means, stdevs = scale(data_matrix)

    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

Dimensionality Reduction

- Sometimes the “actual” (or useful) dimensions of the data might not correspond to the dimensions we have.
- Some of the more popular methods include:
  - Principal Components Analysis
  - Singular Value Decomposition
  - Non-Negative Matrix Factorization

```python
# evaluate logistic regression model on raw data
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=10, n_redundant=10, random_state=7)
# define the model
model = LogisticRegression()
# evaluate model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv,
n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

```
Accuracy: 0.824 (0.034)
```

MACHINE LEARNING

Models

- To creating and using models that are learned from data.
- Typically, our goal will be to use existing data to develop models that we can use to *predict* various outcomes for new data.
  - Predicting whether an email message is spam or not
  - Predicting whether a credit card transaction is fraudulent
  - Predicting which advertisement a shopper is most likely to click on
  - Predicting which football team is going to win the Super Bowl
- *Supervised models* - In which there is a set of data labeled with the correct answers to learn from,
- Unsupervised models - In which there are no such labels.

Defining Models

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results

def train_test_split(x, y, test_pct):
    data = zip(x, y)                                  # pair corresponding values
    train, test = split_data(data, 1 - test_pct)      # split the data set of pairs
    x_train, y_train = zip(*train)                    # magical un-zip trick
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test

model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

Checking Correctness

- Given a set of labeled data and such a predictive model, every data point lies in one of four categories:
  - True positive: “This message is spam, and we correctly predicted spam.”
  - False positive (Type 1 Error): “This message is not spam, but we predicted spam.”
  - False negative (Type 2 Error): “This message is spam, but we predicted not spam.”
  - True negative: “This message is not spam, and we correctly predicted not spam.”

	Spam	not Spam
predict “Spam”	True Positive	False Positive
predict “Not Spam”	False Negative	True Negative

	leukemia	no leukemia	total
“Luke”	70	4,930	5,000
not “Luke”	13,930	981,070	995,000
total	14,000	986,000	1,000,000

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)     # 0.98114
```

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)    # 0.014
```

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print recall(70, 4930, 13930, 981070)       # 0.005
```

```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

Simple Linear Regression

- **def** predict(alpha, beta, x_i):

  return beta * x_i + alpha

- **def** error(alpha, beta, x_i, y_i):

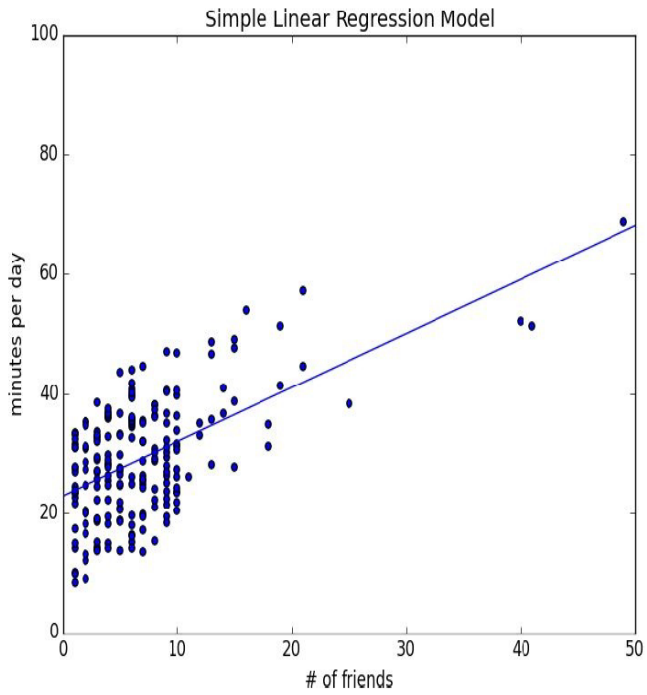
  *"""the error from predicting beta * x_i + alpha when the actual value is y_i"""*

  return y_i - predict(alpha, beta, x_i)

- **def** sum_of_squared_errors(alpha, beta, x, y):

  return sum(error(alpha, beta, x_i, y_i) ** 2

  for x_i, y_i **in** zip(x, y))



- **def** least_squares_fit(x, y):

  """given training values for x and y,find the least-squares values of alpha and beta"""

  beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)

  alpha = mean(y) - beta * mean(x)

  return alpha, beta

- alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)

References

- Python Homepage
  - http://www.python.org
- Python Tutorial
  - http://docs.python.org/tutorial/
- Python Documentation
  - http://www.python.org/doc
- Python Library References
  - http://docs.python.org/release/2.5.2/lib/lib.html
- Python Add-on Packages:
  - http://pypi.python.org/pypi
- Data Science from Scratch – First principles with Python, Joel Grus, O'Reilly.
- Python Data Science Handbook – Essential Tools for Working with Data, Jake VanderPlas, O'Reilly.