Lab Session 1 - Python

> Your name:

1. Objectives

You have to work alone for this lab session. This introduction to python (Lab_Session*1*-_Python.ipynb) will not be graded. Therefore, it is not compulsory. If you already have experience with python, you can start with the next notebook.

The objective of this lab session is to make you familiar with Jupyter and Python. The lab is based on Google's Python Class and Machine Learning in Action by P. Harrington.

After giving the set-up information in Section 2, Jupyter is shortly introduced in Section 3 and it is explained why Python was chosen for this course in Section 4. This lab session continues with (guided) exercises on strings, lists, sorting and files to make you familiar with Python. After this lab session, you should be able to write applications using Python.

2. Setup

Python is free and open source, available for all operating systems from python.org (http://python.org/).

However, we will install Anaconda (https://www.continuum.io/downloads). Anaconda conveniently installs Python, the Jupyter Notebook, and other commonly used packages for scientific computing and data science.

Follow Anaconda's instructions for downloading and installing the Python 2.7 version. You can also use python 3.x, however the labs were created using python 2.7. Using python 3.x should only give minor changes.

Once Anaconda is installed, run the following command in the Command Prompt or Terminal to install Jupyter:

```
conda install jupyter
```

3. Jupyter

The Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. The notebooks of this course are all associated with the IPython kernel, and therefore run Python code.

Starting the notebook server using the command line

You can start the notebook server from the command line (Terminal on Mac/Linux, CMD prompt on Windows) by running the following command:

```
jupyter notebook
```

This will print some information about the notebook server in your command prompt or terminal, including the URL of the web application (by default, http://localhost:8888 (http://localhost:8888)). It will then open your default web browser to this URL.

When the notebook opens, you will see the notebook dashboard, which will show a list of the notebooks, files, and subdirectories in the directory where the notebook server was started. So you will want to start a notebook server in the highest directory in your filesystem where your notebooks can be found.

When starting a notebook server from the command line, you can also open a particular notebook directly, bypassing the dashboard, with:

```
jupyter notebook my_notebook.ipynb
```

The .ipynb extension is assumed if no extension is given.

Code cells allow you to enter and run code

Run a code cell using `Shift-Enter` or pressing the ▢ button in the toolbar above:

In [2]:

```
a = 10
```

In [ ]:

```
print(a)
```

You can also use the Alt-Enter keyboard shortcut to run the current cell and insert a new one below.

Basic workflow

The normal workflow in a lab notebook is as follows:

Typically, you will work on a lab session in pieces, organizing related ideas into cells and moving forward once previous parts work correctly.

At certain moments, it may be necessary to interrupt a calculation which is taking too long to complete. This may be done with the *Kernel* | *Interrupt* menu option, or the *Ctrl-m i* keyboard shortcut. Similarly, it may be necessary or desirable to restart the whole computational process, with the *Kernel* | *Restart* menu option or *Ctrl-m .* shortcut.

A notebook may be downloaded in either a .ipynb or .py file from the menu option File | Download as. Choosing the .py option downloads a Python .py script, in which all rich output has been removed and the content of markdown cells have been inserted as comments.

3. Why Python?

Python is a great language for machine learning for a large number of reasons. First, Python has clear syntax. Second, it makes text manipulation extremely easy. A large number of people and organizations use Python, so there are many development samples and a lot of documentation.

The clear syntax of Python has earned it the name executable pseudo-code. The default install of Python already carries high-level data types like lists, tuples, dictionaries, sets, queues, and so on, which you don't have to program in yourself. These high-level data types make abstract concepts easy to implement. With Python, you can program in any style you're familiar with: object-oriented, procedural, functional, and so on.

With Python it's easy to process and manipulate text, which makes it ideal for processing non-numeric data. You can get by in Python with little to no regular expression usage. There are a number of libraries for using Python to access web pages, and the intuitive text manipulation makes it easy to extract data from HTML.

Python is popular in the scientific and financial communities as well. A number of scientific libraries such as SciPy and NumPy allow you to do vector and matrix operations. This makes the code even more readable and allows you to write code that looks like linear algebra. In addition, the scientific libraries SciPy and NumPy are compiled using lower-level languages (C and Fortran); this makes doing computations with these tools much faster. The scientific tools in Python work well with a plotting tool called Matplotlib. Matplotlib can plot 2D and 3D and can handle most types of plots commonly used in the scientific world.

4. Python Introduction

Python is a dynamic, interpreted language. Source code does not declare the types of variables or parameters or methods -- just assign to them and go. This makes the code short and flexible, and you lose the compile-time type checking in the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs. It raises a runtime error if the code tries to read from a variable that has not been given a value. An excellent way to see how Python code works is to run the code below. Like C++ and Java, Python is case sensitive so "a" and "A" are different variables. The end of a line marks the end of a statement, so unlike C++ and Java, Python does not require a semicolon at the end of each statement. Comments begin with a '#' and extend to the end of the line.

In [ ]:

```
a = 6     # set a variable
a         # entering an expression prints its value
```

In [ ]:

```
a+2
```

In [ ]:

```
a = 'hi'  # a can hold a string just as well
a
```

In [ ]:

```
len(a)    # call the len() function on a string
```

In [ ]:

```
foo(a)    # try something that doesn't work
```

Functions

Functions in Python are defined like this:

In [17]:

```
# Defines a "repeat" function that takes 2 arguments.
def repeat(s, exclaim):
    """Returns the string s repeated 3 times.
    If exclaim is true, add exclamation marks.
    """
    result = s + s + s # can also use "s * 3" which is faster (Why?)
    if exclaim:
        result = result + '!!'
    return result
```

Notice how the lines that make up the function or if-statement are grouped by all having the same level of indentation. Two different ways to repeat strings are presented, using the + operator which is more user-friendly, but *also works because it is Python*'s "*repeat*" *operator, meaning that* '-' 10 gives '----------', a neat way to create an onscreen "line". As hinted in the code, *works faster than +, the reason being that* calculates the size of the resulting object once whereas with +, that calculation is made each time + is called. Both + and * are called "overloaded" operators because they mean different things for numbers vs. for strings (and other data types).

The "def" defines the function with its parameters within parentheses and its code indented. The first line of a function can be a documentation string ("docstring") that describes what the function does. The docstring can be a single line, or a multi-line description as in the example above. Variables defined in the function are local to that function, so the "result" in the above function is separate from a "result" variable in another function. The return statement can take an argument, in which case that is the value returned to the caller.

Here is code that calls the above repeat() function, printing what it returns:

In [ ]:

```
repeat('Yay', False)
```

In [ ]:

```
repeat('Woo Hoo', True)
```

Indentation

One unusual Python feature is that the whitespace indentation of a piece of code affects its meaning. A logical block of statements such as the ones that make up a function should all have the same indentation, set in from the indentation of their parent function or "if" or whatever. If one of the lines in a group has a different indentation, it is flagged as a syntax error.

Variable names

Since Python variables don't have any type spelled out in the source code, it is extra helpful to give meaningful names to your variables to remind yourself of what is going on. As many basic Python errors result from forgetting what type of value is in each variable, use "name" if it's a single name, and "names" if it is a list of names, and "tuples" if it is a list of tuples.

Modules and Imports

One file of Python code is called a module. The file "test.py" is also known as the module "test". A module essentially contains variable definitions like, "x = 6" and "def foo()". Suppose the file "test.py" contains a "def foo()". The fully qualified name of that foo function is "test.foo". In this way, various Python modules can name their functions and variables whatever they want, and the variable names won't conflict -- module1.foo is different from module2.foo.

With the statement "import xxx" you can access the definitions in the xxx module/package and makes them available by their fully-qualified name, e.g. xxx.function().

Online help and dir

There are a variety ways to get help for Python.

- Do a Google search, starting with the word "python", like "python list" or "python string lowercase". The first hit is often the answer.
- The official Python docs site -- docs.python.org -- has high quality docs. Nonetheless, a Google search of a couple words is often quicker.
- There is also an official Tutor mailing list specifically designed for those who are new to Python and/or programming.
- Many questions (and answers) can be found on StackOverflow.
- Use the help() [and dir()] functions described below.

Inside the Python interpreter, the help() function pulls up documentation strings for various modules, functions, and methods. These doc strings are similar to Java's javadoc. This is one way to get quick access to docs. Here are some ways to call help() from inside the interpreter:

- help(len) -- docs for the built in len function (note here you type "len" not "len()" which would be a call to the function)
- help(list) -- docs for the built in "list" module
- help(list.append) -- docs for the append() function in the list module

5. Python Strings

Python has a built-in string class named "str" with many handy features (there is an older module named "string"). String literals can be enclosed by either double or single quotes, although single quotes are more commonly used. Backslash escapes work the usual way within both single and double quoted literals -- e.g. \n \' \". A double quoted string literal can contain single quotes without any fuss (e.g. "I didn't do it") and likewise single quoted string can contain double quotes. A string literal can span multiple lines, but there must be a backslash \ at the end of each line to escape the newline. String literals inside triple quotes, """ or ''', can contain multiple lines of text.

Python strings are "immutable" which means they cannot be changed after they are created. Since strings can't be changed, we construct *new* strings as we go to represent computed values. So for example the expression ('hello' + 'there') takes in the 2 strings 'hello' and 'there' and builds a new string 'hellothere'.

Characters in a string can be accessed using the standard [ ] syntax, and like Java and C++, Python uses zero-based indexing, so if str is 'hello' str[1] is 'e'. If the index is out of bounds for the string, Python halts and raises an error.

The len(string) function returns the length of a string. The [ ] syntax and the len() function actually work on any sequence type -- strings, lists, etc.. Python tries to make its operations work consistently across different types. The '+' operator can concatenate two strings. Notice in the code below that variables are not pre-declared -- just assign to them and go.

In [22]:

```
s = 'hi'
```

In [ ]:

```
s[1]
```

In [ ]:

```
len(s)
```

In [ ]:

```
s + ' there'
```

Unlike Java, the '+' does not automatically convert numbers or other types to string form. The str() function converts values to a string form so they can be combined with other strings.

In [26]:

```
pi = 3.14
```

In [ ]:

```
text = 'The value of pi is ' + pi        # Does not work
```

In [ ]:

```
text = 'The value of pi is ' + str(pi)   # Does work
```

For numbers, the standard operators, +, /, * work in the usual way. There is no ++ operator, but +=, -=, etc. work. If you want integer division, it is most correct to use 2 slashes -- e.g. 6 // 5 is 1.

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result. Here are some of the most common string methods:

- s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
- s.strip() -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
- s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

A google search for "python str" should lead you to the official python.org string methods which lists all the str methods.

Python does not have a separate character type. Instead an expression like s[8] returns a string-length-1 containing the character. With that string-length-1, the operators ==, <=, ... all work as you would expect, so mostly you don't need to know that Python does not have a separate scalar "char" type.

The "slice" syntax is a handy way to refer to sub-parts of sequences -- typically strings and lists. The slice s[start:end] is the elements beginning at start and extending up to but not including end. The standard zero-based index numbers give easy access to chars near the start of the string.Python uses negative numbers in slice to give easy access to the chars at the end of the string: s[-1] is the last char of the string, s[-2] is the next-to-last char, and so on. Thus, negative index numbers count back from the end of the string.

Python has a printf()-like facility to put together a string. The % operator takes a printf-type format string on the left (%d int, %s string, %f/%g floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parenthesis):

In [28]:

```
text = "%d little %s" % (3, 'pigs')
```

Suppose you want to break the above line into separate lines. You cannot just split the as you might in other languages, since by default Python treats each line as a separate statement. To fix this, enclose the whole expression in an outer set of parenthesis -- then the expression is allowed to span multiple lines. This code-across-lines technique works with the various grouping constructs: ( ), [ ], { }.

In [30]:

```
text = ("%d little %s" %
    (3, 'pigs'))
```

Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (:) and indentation/whitespace to group statements. The boolean test for an if does not need to be in parenthesis, and it can have *elif* and *else* clauses.

Any value can be used as an if-test. The "zero" values all count as false: None, 0, empty string, empty list, empty dictionary. There is also a Boolean type with two values: True and False (converted to an int, these are 1 and 0). Python has the usual comparison operations: ==, !=, <, <=, >, >=. Unlike Java and C, == is overloaded to work correctly with strings. The boolean operators are the spelled out words *and*, *or*, *not* (Python does not use the C-style && ||).

Here's what the code might look like for a policeman pulling over a speeder -- notice how each block of then/else statements starts with a : and the statements are grouped by their indentation:

In [ ]:

```
if speed >= 80:
    if mood == 'terrible' or speed >= 100:
      text = 'You have the right to remain silent.'
    elif mood == 'bad' or speed >= 90:
      text = "I'm going to have to write you a ticket."
    else:
      text = "Let's try to keep it under 80 ok?"
```

Exercises

To practice the material in this section, solve the below exercises.

Ex.1. Donuts

Given an int count of a number of donuts, return a string of the form 'Number of donuts: *count*', where *count* is the number passed in. However, if the count is 10 or more, then use the word 'many' instead of the actual count. So donuts(5) returns 'Number of donuts: 5' and donuts(23) returns 'Number of donuts: many'

In [ ]:

```
def donuts(count):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
donuts(4)
```

In [ ]:

```
donuts(9)
```

In [ ]:

```
donuts(10)
```

In [ ]:

```
donuts(99)
```

Ex.2. Both_ends

Given a string s, return a string made of the first 2 and the last 2 chars of the original string, so 'spring' yields 'spng'. However, if the string length is less than 2, return instead the empty string.

In [ ]:

```
def both_ends(s):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
both_ends('spring')
```

In [ ]:

```
both_ends('Hello')
```

In [ ]:

```
both_ends('a')
```

In [ ]:

```
both_ends('xyz')
```

Ex.3. Fix_start

Given a string s, return a string where all occurences of its first char have been changed to ' * ', except do not change the first char itself. E.g. 'babble' yields 'ba**le'. Assume that the string is length 1 or more.

Hint: s.replace(stra, strb) returns a version of string s where all instances of stra have been replaced by strb.

In [ ]:

```
def fix_start(s):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
fix_start('babble')
```

In [ ]:

```
fix_start('aardvark')
```

In [ ]:

```
fix_start('google')
```

In [ ]:

```
fix_start('donut')
```

Ex.4. MixUp

Given strings a and b, return a single string with a and b separated by a space 'a b', except swap the first 2 chars of each string.
E.g. 'mix', 'pod' -> 'pox mid'
'dog', 'dinner' -> 'dig donner'
Assume a and b are length 2 or more.

In [ ]:

```
def mix_up(a, b):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
mix_up('mix', 'pod')
```

In [ ]:

```
mix_up('dog', 'dinner')
```

In [ ]:

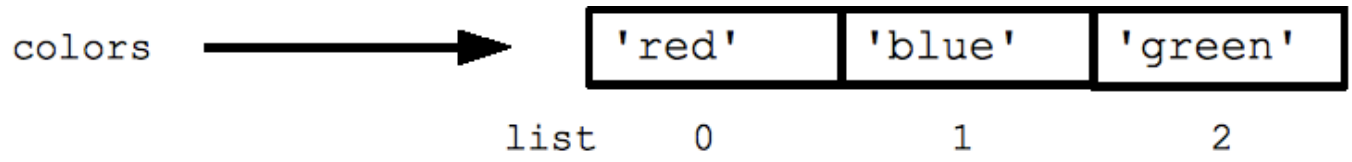
```
mix_up('gnash', 'sport')
```

In [ ]:

```
mix_up('pezzy', 'firm')
```

6. Python Lists

Python has a great built-in list type named "list". List literals are written within square brackets [ ]. Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0.



In [21]:

```
colors = ['red', 'blue', 'green']
```

In [ ]:

```
colors[0]
```

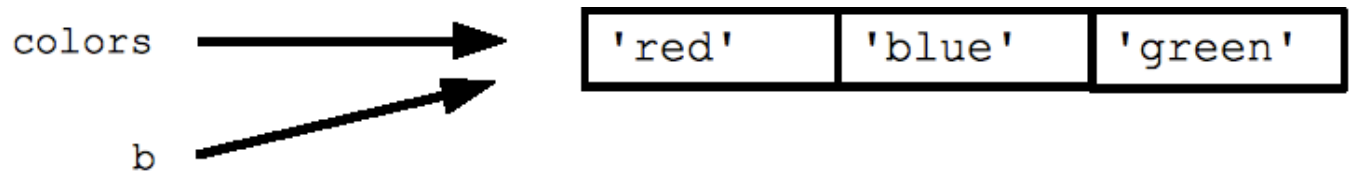
In [ ]:

```
colors[2]
```

In [ ]:

```
len(colors)
```

Assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.



In [ ]:

```
b = colors
```

The "empty list" is just an empty pair of brackets [ ]. The '+' works to append two lists, so [1, 2] + [3, 4] yields [1, 2, 3, 4] (this is just like + with strings).

FOR and IN

Python's *for* and *in* constructs are extremely useful, e.g. for lists. The *for* construct -- for var in list -- is an easy way to look at each element in a list (or other collection).

In [ ]:

```
squares = [1, 4, 9, 16]
sum = 0
for num in squares:
  sum += num
sum
```

If you know what sort of thing is in the list, use a variable name in the loop that captures that information such as "num", or "name", or "url". Since python code does not have other syntax to remind you of types, your variable names are a key way for you to keep straight what is going on.

The *in* construct on its own is an easy way to test if an element appears in a list (or other collection) -- value in collection -- tests if the value is in the collection, returning True/False.

In [ ]:

```
list = ['larry', 'curly', 'moe']
if 'curly' in list:
   print('yay')
```

The for/in constructs are very commonly used in Python code and work on data types other than list.

You can also use for/in to work on a string. The string acts like a list of its chars, so for ch in s: print ch prints all the chars in a string.

Range

The range(n) function yields the numbers 0, 1, ... n-1, and range(a, b) returns a, a+1, ... b-1 -- up to but not including the last number. The combination of the for-loop and the range() function allow you to build a traditional numeric for loop:

In [ ]:

```
## print the numbers from 0 through 99
for i in range(100):
    print(i)
```

There is a variant xrange() which avoids the cost of building the whole list for performance sensitive cases (in Python 3000, range() will have the good performance behavior and you can forget about xrange()).

While Loop

Python also has the standard while-loop, and the *break* and *continue* statements work as in C++ and Java, altering the course of the innermost loop. The above for/in loops solves the common case of iterating over every element in a list, but the while loop gives you total control over the index numbers. Here's a while loop which accesses every 3rd element in a list:

```
## Access every 3rd element in a list
i = 0
while i < len(a):
    print(a[i])
    i = i + 3
```

List Methods

Here are some other common list methods.

- list.append(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
- list.sort() -- sorts the list in place (does not return it).
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Notice that these are *methods* on a list object, while len() is a function that takes the list (or string or whatever) as an argument.

In [36]:

```
list = ['larry', 'curly', 'moe']
list.append('shemp')         ## append elem at end
list.insert(0, 'xxx')        ## insert elem at index 0
list.extend(['yyy', 'zzz'])  ## add list of elems at end
```

In [ ]:

```
list
```

In [ ]:

```
list.index('curly')
```

In [ ]:

```
list.remove('curly')         ## search and remove that element
list.pop(1)                  ## removes and returns 'larry'
list
```

Note that the above methods do not *return* the modified list, they just modify the original list.

In [ ]:

```
list = [1, 2, 3]
print(list.append(4))        ## Does not work, append() returns None
```

In [ ]:

```
## Correct pattern:
list.append(4)
print(list)
```

List Build Up

One common pattern is to start a list a the empty list [], then use append() or extend() to add elements to it:

In [ ]:

```
list = []          ## Start as the empty list
list.append('a')   ## Use append() to add elements
list.append('b')
```

List Slices

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

In [ ]:

```
list = ['a', 'b', 'c', 'd']
print(list[1:-1])       ## ['b', 'c']
list[0:2] = 'z'         ## replace ['a', 'b'] with ['z']
print(list)             ## ['z', 'c', 'd']
```

Exercises

To practice the material in this section, solve the below exercises.

Ex.5. match_ends

Given a list of strings, return the count of the number of strings where the string length is 2 or more and the first and last chars of the string are the same.
Note: python does not have a ++ operator, but += works.

In [ ]:

```
def match_ends(words):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
match_ends(['aba', 'xyz', 'aa', 'x', 'bbb'])
```

In [ ]:

```
match_ends(['', 'x', 'xy', 'xyx', 'xx'])
```

In [ ]:

```
match_ends(['aaa', 'be', 'abc', 'hello'])
```

7. Python Sorting

The easiest way to sort is with the sorted(list) function, which takes a list and returns a new list with those elements in sorted order. The original list is not changed.

In [ ]:

```
a = [5, 1, 4, 3]
print(sorted(a))
print(a)
```

It's most common to pass a list into the sorted() function, but in fact it can take as input any sort of iterable collection. The older list.sort() method is an alternative detailed below. The sorted() function can be customized though optional arguments. The sorted() optional argument reverse=True, e.g. sorted(list, reverse=True), makes it sort backwards.

In [ ]:

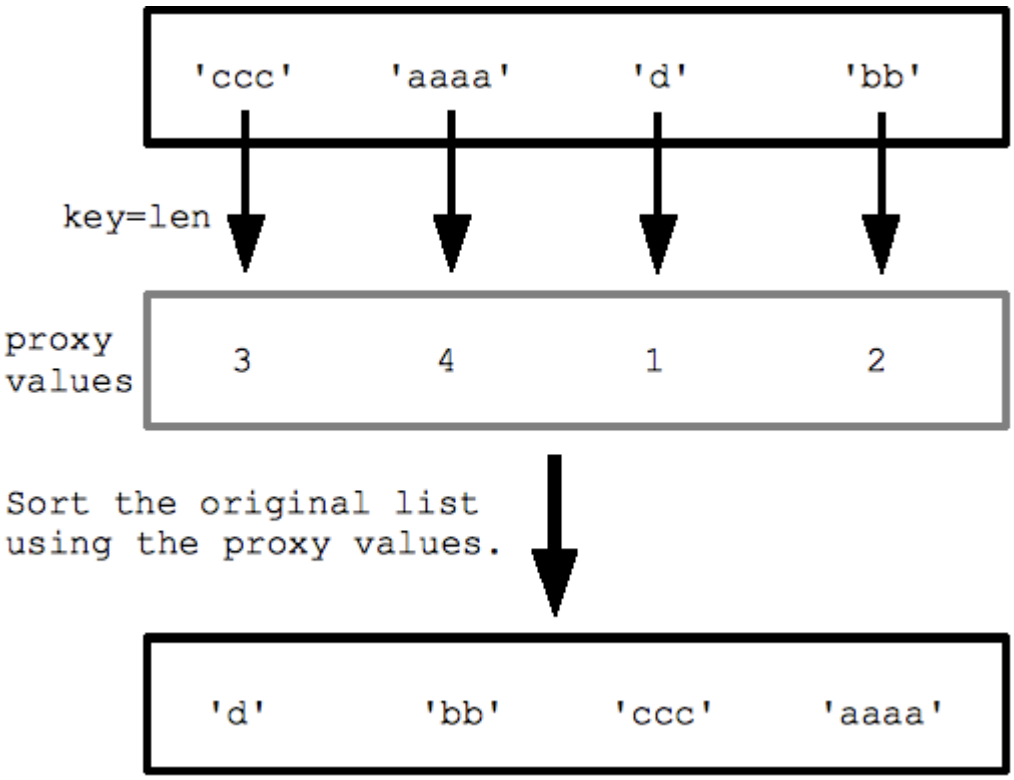
```
strs = ['aa', 'BB', 'zz', 'CC']
print(sorted(strs))                ## case sensitive sorting
print(sorted(strs, reverse=True))
```

Custom Sorting With key=

For more complex custom sorting, sorted() takes an optional "key=" specifying a "key" function that transforms each element before comparison. The key function takes in 1 value and returns 1 value, and the returned "proxy" value is used for the comparisons within the sort. For example with a list of strings, specifying key=len (the built in len() function) sorts the strings by length, from shortest to longest. The sort calls len() for each string to get the list of proxy length values, and the sorts with those proxy values.

In [ ]:

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print(sorted(strs, key=len))
```



As another example, specifying "str.lower" as the key function is a way to force the sorting to treat uppercase and lowercase the same:

In [ ]:

```
## "key" argument specifying str.lower function to use for sorting
sorted(strs, key=str.lower)
```

You can also pass in your own MyFn as the key function, like this:

In [ ]:

```
## Say we have a list of strings we want to sort by the last letter of the strin
g.
strs = ['xc', 'zb', 'yd' ,'wa']

## Write a little function that takes a string, and returns its last letter.
## This will be the key function (takes in 1 value, returns 1 value).
def MyFn(s):
   return s[-1]

## Now pass key=MyFn to sorted() to sort by the last letter:
print(sorted(strs, key=MyFn))
```

To use key= custom sorting, remember that you provide a function that takes one value and returns the proxy value to guide the sorting. There is also an optional argument "cmp=cmpFn" to sorted() that specifies a traditional two-argument comparison function that takes two values from the list and returns negative/0/positive to indicate their ordering. The built in comparison function for strings, ints, ... is cmp(a, b), so often you want to call cmp() in your custom comparator. The newer one argument key= sorting is generally preferable.

sort() method

As an alternative to sorted(), the sort() method on a list sorts that list into ascending order, e.g. list.sort(). The sort() method changes the underlying list and returns None, so use it like this:

```
alist.sort()          ## correct
alist = blist.sort() ## incorrect, sort() returns None
```

The above is a very common misunderstanding with sort() -- it *does not return* the sorted list. The sort() method must be called on a list; it does not work on any enumerable collection (but the sorted() function above works on anything). The sort() method predates the sorted() function, so you will likely see it in older code. The sort() method does not need to create a new list, so it can be a little faster in the case that the elements to sort are already in a list.

Tuples

A tuple is a fixed size grouping of elements, such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable and do not change size (tuples are not strictly immutable since one of the contained elements could be mutable). Tuples play a sort of "struct" role in Python -- a convenient way to pass around a little logical, fixed size bundle of values. A function that needs to return multiple values can just return a tuple of the values. For example, if I wanted to have a list of 3-d coordinates, the natural python representation would be a list of tuples, where each tuple is size 3 holding one (x, y, z) group.

To create a tuple, just list the values within parenthesis separated by commas. The "empty" tuple is just an empty pair of parenthesis. Accessing the elements in a tuple is just like a list -- len(), [ ], for, in, etc. all work the same.

In [ ]:

```
tuple = (1, 2, 'hi')
print(len(tuple))       ## 3
print(tuple[2])         ## hi
tuple[2] = 'bye'        ## NO, tuples cannot be changed
tuple = (1, 2, 'bye')   ## this works
```

To create a size-1 tuple, the lone element must be followed by a comma.

In [ ]:

```
tuple = ('hi',)        ## size-1 tuple
```

The comma is necessary to distinguish the tuple from the ordinary case of putting an expression in parentheses. In some cases you can omit the parenthesis and Python will see from the commas that you intend a tuple.

Assigning a tuple to an identically sized tuple of variable names assigns all the corresponding values. If the tuples are not the same size, it throws an error. This feature works for lists too.

In [ ]:

```
(x, y, z) = (42, 13, "hike")
print(z)
```

Exercises

To practice the material, solve the problems below:

Ex.6. front_x

Given a list of strings, return a list with the strings in sorted order, except group all the strings that begin with 'x' first.
E.g. ['mix', 'xyz', 'apple', 'xanadu', 'aardvark'] yields ['xanadu', 'xyz', 'aardvark', 'apple', 'mix']
Hint: this can be done by making 2 lists and sorting each of them before combining them.

In [ ]:

```
def front_x(words):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
front_x(['bbb', 'ccc', 'axx', 'xzz', 'xaa'])
```

In [ ]:

```
front_x(['ccc', 'bbb', 'aaa', 'xcc', 'xaa'])
```

In [ ]:

```
front_x(['mix', 'xyz', 'apple', 'xanadu', 'aardvark']
```

Ex.7. sort_last

Given a list of non-empty tuples, return a list sorted in increasing order by the last element in each tuple.
E.g. [(1, 7), (1, 3), (3, 4, 5), (2, 2)] yields [(2, 2), (1, 3), (3, 4, 5), (1, 7)]
Hint: use a custom key= function to extract the last element form each tuple.

In [ ]:

```
def sort_last(tuples):
    # +++your code here+++
    return
```

Test your code:

In [ ]:

```
sort_last([(1, 3), (3, 2), (2, 1)])
```

In [ ]:

```
sort_last([(2, 3), (1, 2), (3, 1)])
```

In [ ]:

```
sort_last([(1, 7), (1, 3), (3, 4, 5), (2, 2)])
```

8. Python Dict and File

Dict Hash Table

Python's efficient key/value hash table structure is called a "dict". The contents of a dict can be written as a series of key:value pairs within braces { }, e.g. dict = {key1:value1, key2:value2, ... }. The "empty dict" is just an empty pair of curly braces {}.

Looking up or setting a value in a dict uses square brackets, e.g. dict['foo'] looks up the value under the key 'foo'. Strings, numbers, and tuples work as keys, and any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable). Looking up a value which is not in the dict throws a KeyError -- use "in" to check if the key is in the dict, or use dict.get(key) which returns the value or None if the key is not present (or get(key, not-found) allows you to specify what value to return in the not-found case).

In [ ]:

```
## Can build up a dict by starting with the the empty dict {}
## and storing key/value pairs into the dict like this:
## dict[key] = value-for-that-key
dict = {}
dict['a'] = 'alpha'
dict['g'] = 'gamma'
dict['o'] = 'omega'

print(dict)
```

In [ ]:

```
print(dict['a'])       ## Simple lookup
```

In [ ]:

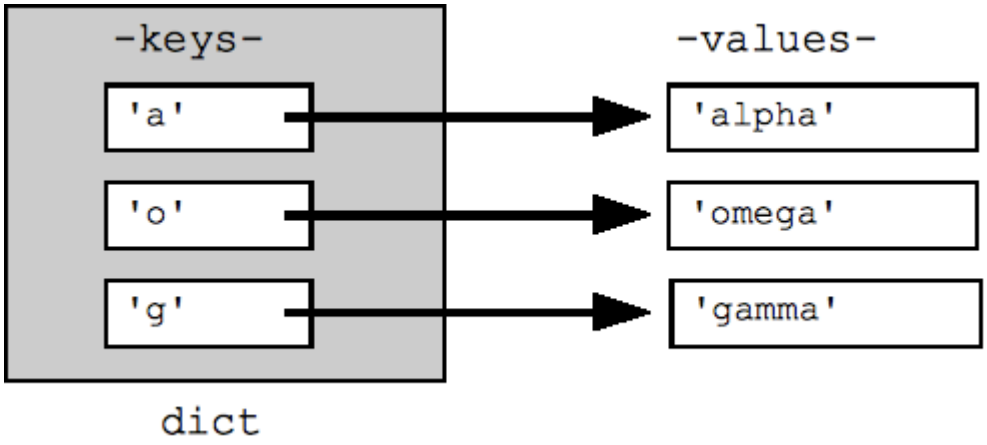
```
dict['a'] = 6          ## Put new key/value into dict
'a' in dict
```

In [ ]:

```
print(dict['z'])     ## Throws KeyError
```

In [ ]:

```
if 'z' in dict: print(dict['z'])  ## Avoid KeyError
print(dict.get('z'))              ## None (instead of KeyError)
```



A for loop on a dictionary iterates over its keys by default. The keys will appear in an arbitrary order. The methods dict.keys() and dict.values() return lists of the keys or values explicitly. There's also an items() which returns a list of (key, value) tuples, which is the most efficient way to examine all the key value data in the dictionary. All of these lists can be passed to the sorted() function.

In [ ]:

```
## By default, iterating over a dict iterates over its keys.
## Note that the keys are in a random order.

for key in dict: print(key)
```

In [ ]:

```
## Exactly the same as above

for key in dict.keys(): print(key)
```

In [ ]:

```
## Get the .keys() list:

print(dict.keys())
```

In [ ]:

```
## Likewise, there's a .values() list of values

print(dict.values())
```

In [ ]:

```
## Common case -- loop over the keys in sorted order, accessing each key/value

for key in sorted(dict.keys()):
    print(key, dict[key])
```

In [ ]:

```
## .items() is the dict expressed as (key, value) tuples

print(dict.items())
```

In [ ]:

```
## This loop syntax accesses the whole dict by looping over the .items() tuple l
ist, accessing one (key, value)
## pair on each iteration.

for k, v in dict.items(): print(k, '>', v)
```

There are "iter" variants of these methods called iterkeys(), itervalues() and iteritems() which avoid the cost of constructing the whole list -- a performance win if the data is huge.

From a performance point of view, the dictionary is one of the greatest tools as an easy way to organize data. For example, you might read a log file where each line begins with an ip address, and store the data into a dict using the ip address as the key, and the list of lines where it appears as the value. Once you've read in the whole file, you can look up any ip address and instantly see its list of lines. The dictionary takes in scattered data and make it into something coherent.

Dict Formatting

The % operator works conveniently to substitute values from a dict into a string by name:

In [18]:

```
hash = {}
hash['word'] = 'garfield'
hash['count'] = 42
s = 'I want %(count)d copies of %(word)s' % hash  # %d for int, %s for string
```

In [ ]:

```
print(s)
```

Del

The "del" operator does deletions. In the simplest case, it can remove the definition of a variable, as if that variable had not been defined. Del can also be used on list elements or slices to delete that part of the list and to delete entries from a dictionary.

In [ ]:

```
var = 6
del var         ## var no more!

list = ['a', 'b', 'c', 'd']
del list[0]     ## Delete first element
del list[-2:]   ## Delete last two elements
print(list)

dict = {'a':1, 'b':2, 'c':3}
del dict['b']   ## Delete 'b' entry
print(dict)
```

Files

The open() function opens and returns a file handle that can be used to read or write a file in the usual way. The code f = open('name', 'r') opens the file into the variable f, ready for reading operations, and use f.close() when finished. Instead of 'r', use 'w' for writing, and 'a' for append. The special mode 'rU' is the "Universal" option for text files where it's smart about converting different line-endings so they always come through as a simple '\n'. The standard for-loop works for text files, iterating through the lines of the file (this works only for text files, not binary files). The for-loop technique is a simple and efficient way to look at all the lines in a text file:

In [ ]:

```
# Echo the contents of a file
f = open('foo.txt', 'r')
for line in f:             ## iterates over the lines of the file
    print(line ,)            ## trailing , so print does not add an end-of-line
 char
                               ## since 'line' already includes the end-of line.
f.close()
```

Reading one line at a time has the nice quality that not all the file needs to fit in memory at one time -- handy if you want to look at every line in a 10 gigabyte file without using 10 gigabytes of memory. The f.readlines() method reads the whole file into memory and returns its contents as a list of its lines. The f.read() method reads the whole file into a single string, which can be a handy way to deal with the text all at once, such as with regular expressions.

For writing, f.write(string) method is the easiest way to write data to an open output file. Or you can use "print" with an open file, but the syntax is nasty: "print >> f, string".

Files Unicode

The "codecs" module provides support for reading a unicode file.

```
import codecs

f = codecs.open('foo.txt', 'rU', 'utf-8')
for line in f:
      # here line is a *unicode* string
```

For writing, use f.write() since print does not fully support unicode.

Incremental Development

When building a Python program, don't write the whole thing in one step. Instead identify just a first milestone, e.g. "the first step is to extract the list of words." Write the code to get to that milestone, and just print your data structures at that point, and then you can do a sys.exit(0) so the program does not run ahead into its not-done parts. Once the milestone code is working, you can work on code for the next milestone. Being able to look at the printout of your variables at one state can help you think about how you need to transform those variables to get to the next state. Python is very quick with this pattern, allowing you to make a little change and run the program to see how it works. Take advantage of that quick turnaround to build your program in little steps.

Exercises

In this exercise, all the basic Python material -- strings, lists, dicts, tuples, files – will be combined.

Ex.8. print_words & print_top

Implement a **print_words(filename)** function that counts how often each word appears in the text and prints:
word1 count1
word2 count2
...

Print the above list in order sorted by word (python will sort punctuation to come before letters -- that's fine).

Store all the words as lowercase, so 'The' and 'the' count as the same word.

Implement a **print_top(filename)** which is similar to print_words() but which prints just the top 20 most common words sorted so the most common word is first, then the next most common, and so on.

Use str.split() (no arguments) to split on all whitespace.

Optional: define a helper function that reads a file and builds and returns a word/count dict for it to avoid code duplication inside print_words() and print_top().

In [ ]:

```
## +++your code here+++
## Define optional helper function
```

In [ ]:

```
## +++your code here+++
## Define print_words(filename) function
```

In [ ]:

```
## +++your code here+++
## Define print_top(filename) function
```

Test your code:

In [ ]:

```
## +++your test code here+++
```

9. NumPy

Officially, NumPy is a matrix type for Python, and a large number of functions to operate on these matrices. Unofficially, it's a library that makes doing calculations easy and faster to execute, because the calculations are done in C rather than Python. Despite the claim that it's a matrix library, there are actually two fundamental data types in NumPy: the array and the matrix. The operations on arrays and matrices are slightly different. If you're familiar with MATLAB™, then the matrix will be most familiar to you. Both types allow you to remove looping operators you'd have to have using only Python.

You simply have to write at the beginning of your script:

In [1]:

```
import numpy
from numpy import *
from numpy.linalg import *
```

You can get some documentation on the module by using the help() function:

In [ ]:

```
help(numpy)
```

Here's an example of things you can do with arrays:

In [ ]:

```
from numpy import array
mm=array((1, 1, 1))
pp=array((1, 2, 3))
pp+mm
```

That would have required a for loop in regular Python. Here are some more operations that would require a loop in regular Python:

- Multiply every number by a constant 2:

In [ ]:

```
pp*2
```

- Square every number:

In [ ]:

```
pp**2
```

You can now access the elements in the array like it was a list:

In [ ]:

```
pp[1]
```

You can also have multidimensional arrays:

In [ ]:

```
jj = array([[1, 2, 3], [1, 1, 1]])
```

These can also be accessed like lists:

In [ ]:

```
jj[0]
```

In [ ]:

```
jj[0][1]
```

You can also access the elements like a matrix:

In [ ]:

```
jj[0,1]
```

When you multiply two arrays together, you multiply the elements in the first array by the elements in the second array:

In [ ]:

```
a1=array([1, 2,3])
a2=array([0.3, 0.2, 0.3])
a1*a2
```

Similar to arrays, you need to import matrix or mat from NumPy:

In [ ]:

```
from numpy import mat, matrix
```

The NumPy keyword mat is a shortcut for matrix.

In [ ]:

```
ss = mat([1, 2, 3])
ss
```

In [ ]:

```
mm = matrix([1, 2, 3])
mm
```

You can access the individual elements of a matrix like this:

In [ ]:

```
mm[0, 1]
```

You can convert Python lists into NumPy matrices:

In [ ]:

```
pyList = [5, 11, 1605]
mat(pyList)
```

Now let's try to multiply two matrices together:

In [ ]:

```
mm*ss
```

That causes an error and won't be done. The matrix datatype enforces the mathematics of matrix operations. You can't multiply a 1x3 matrix by a 1x3 matrix; the inner numbers must match. One of the matrices will need to be transposed so you can multiply a 3x1 and a 1x3 matrix or a 1x3 and a 3x1 matrix. The NumPy matrix data type has a transpose method, so you can do this multiplication quite easily:

In [ ]:

```
mm*ss.T
```

We took the transpose of ss with the .T method. Knowing the dimensions is helpful when debugging alignment errors. If you want to know the dimensions of an array or matrix, you can use the shape function in NumPy:

In [ ]:

```
from numpy import shape
shape(mm)
```

What if you wanted to multiply every element in matrix mm by every element in ss? This is known as element-wise multiplication and can be done with the NumPy multiply function:

In [ ]:

```
from numpy import multiply
multiply(mm, ss)
```

The matrix and array data types have a large number of other useful methods available such as sorting:

In [ ]:

```
mm.sort()
mm
```

Be careful; this method does sort in place, so if you want to keep the original order of your data, you must make a copy first. You can also use the argsort() method to give you the indices of the matrix if a sort were to happen:

In [ ]:

```
dd=mat([4, 5, 1])
dd.argsort()
```

You can also calculate the mean of the numbers in a matrix:

In [ ]:

```
dd.mean()
```

Let's look at multidimensional arrays for a second:

In [ ]:

```
jj = mat([[1, 2, 3,], [8, 8, 8]])
shape(jj)
```

This is a matrix of shape 2x3; to get all the elements in one row, you can use the colon (:) operator with the row number. For example, to get all the elements in row 1, you'd enter

In [ ]:

```
jj[1,:]
```

You can also specify a range of elements. To get all the elements in row 1, columns 0–1, you'd use the following statement:

In [ ]:

```
jj[1,0:2]
```

The elements of the array are internally stored in a sequence, so you can easily reshape vectors or matrices:

In [ ]:

```
myMatrix = array( [ 1, 2, 3, 4, 5, 6, 7, 8])
myMatrix.shape()
```

In [ ]:

```
myMatrix.reshape(2,4)
```