

# Python for Data Analysis and Scientific Computing

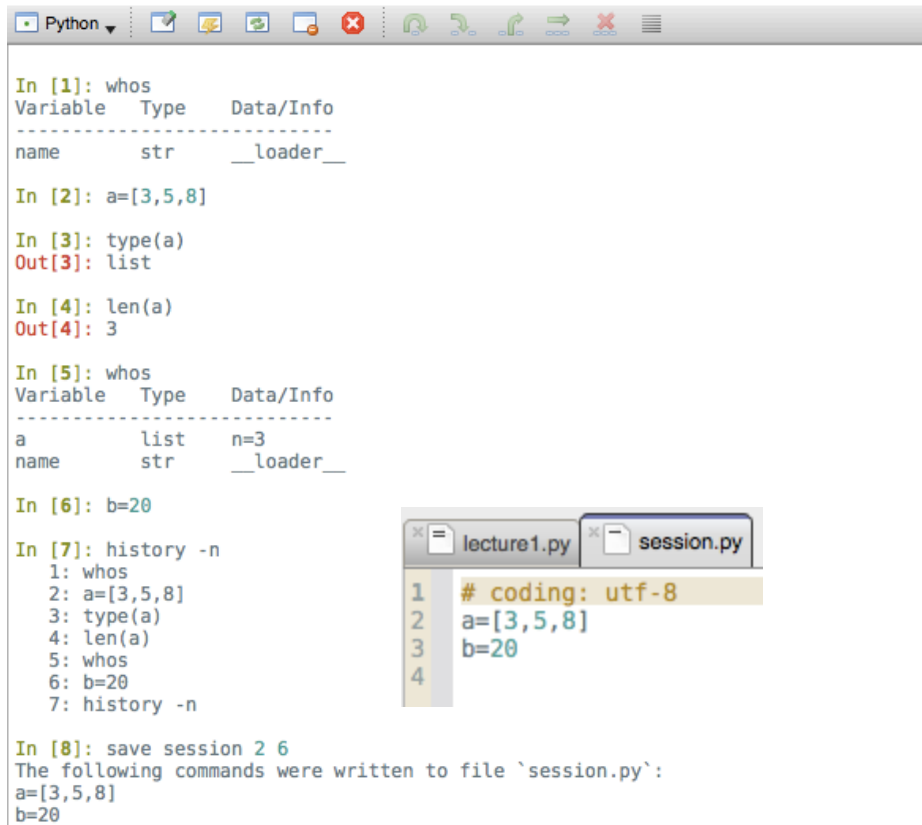
X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

# Python for Data Analysis and Scientific Computing

- Saving individual lines:

`save [option] [log_name [line-region]]` – saving lines of your code that you want to reload next time around



```
In [1]: whos
Variable Type      Data/Info
-----
name      str        __loader__

In [2]: a=[3,5,8]

In [3]: type(a)
Out[3]: list

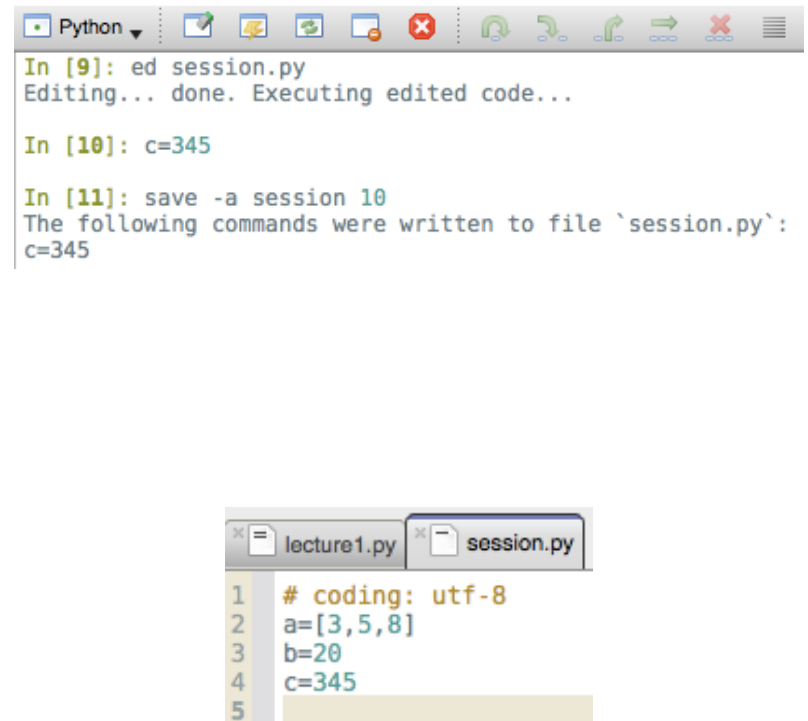
In [4]: len(a)
Out[4]: 3

In [5]: whos
Variable Type      Data/Info
-----
a          list      n=3
name       str        __loader__

In [6]: b=20

In [7]: history -n
1: whos
2: a=[3,5,8]
3: type(a)
4: len(a)
5: whos
6: b=20
7: history -n

In [8]: save session 2 6
The following commands were written to file `session.py`:
a=[3,5,8]
b=20
```



```
In [9]: ed session.py
Editing... done. Executing edited code...

In [10]: c=345

In [11]: save -a session 10
The following commands were written to file `session.py`:
c=345
```

hint: type 'magic' for details

# Python for Data Analysis and Scientific Computing

- Saving your session:

**logstart** [-o|-r|-t] [log\_name [log\_mode]] - Start logging anywhere in a session for the first time with a specific name. defaults is 'ipython\_log.py'. It saves to file 'name' in 'backup' mode. It saves the history up to that point and then continues logging. logstart takes a second optional parameter 'logging mode'. This can be one:

of (note that the modes are given unquoted):

**append** - Keep logging at the end of any existing file

**backup** - Rename any existing file to name~ and start name

**global** - Append to a single logfile in your home directory

**over** - Overwrite any existing log

**rotate** - Create rotating logs: name.1~, name.2~, etc.

**logoff** - Temporarily stop logging. You must have previously started logging

**logon** - Restart logging after temporarily stopped with %logoff

**logstate** - Print the status of the logging system

**logstop** - Fully stop logging and close log file

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages
  - **Numpy** : provides powerful numerical arrays objects, and routines to manipulate them  
<http://www.numpy.org/>
  - **Scipy** : high-level data processing routines. Optimization, regression, interpolation, etc  
<http://www.scipy.org/>
  - **Matplotlib** : 2-D visualization, “publication-ready” plots  
<http://matplotlib.sourceforge.net/>
  - **Mayavi** : 3-D visualization  
<http://code.enthought.com/projects/mayavi/>

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages



## Classification

Identifying to which category an object belongs to.

**Applications:** Spam detection, Image recognition.

**Algorithms:** SVM, nearest neighbors, random forest, ... — Examples

## Regression

Predicting a continuous-valued attribute associated with an object.

**Applications:** Drug response, Stock prices.

**Algorithms:** SVR, ridge regression, Lasso, ... — Examples

## Clustering

Automatic grouping of similar objects into sets.

**Applications:** Customer segmentation, Grouping experiment outcomes

**Algorithms:** k-Means, spectral clustering, mean-shift, ... — Examples

## Dimensionality reduction

Reducing the number of random variables to consider.

**Applications:** Visualization, Increased efficiency

**Algorithms:** PCA, feature selection, non-negative matrix factorization. — Examples

## Model selection

Comparing, validating and choosing parameters and models.

**Goal:** Improved accuracy via parameter tuning

**Modules:** grid search, cross validation, metrics. — Examples

## Preprocessing

Feature extraction and normalization.

**Application:** Transforming input data such as text for use with machine learning algorithms.

**Modules:** preprocessing, feature extraction. — Examples

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

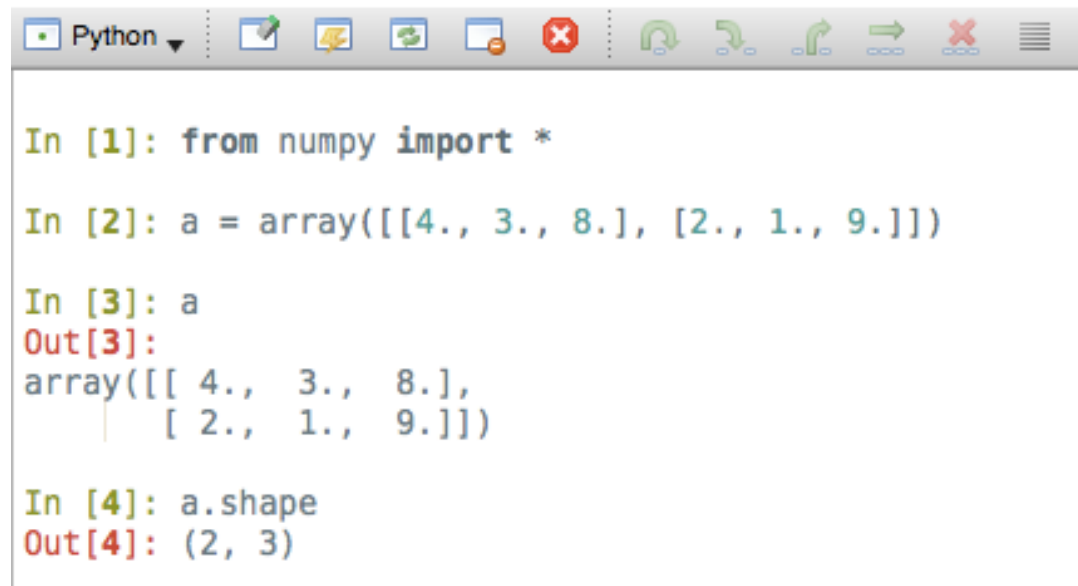
## NumPy :

- This is the fundamental package for scientific computing with Python
- Some of its most attractive key points are:
  - easy **integration with C/C++ and Fortran** code
  - Fourier transform, **linear algebra**, and random number capabilities
  - a powerful **N-dimensional array** object
  - sophisticated (**broadcasting**) functions
  - NumPy can be used as an efficient **multi-dimensional container** of generic data, where **arbitrary data-types can be defined**. This allows NumPy to **easily integrate** with a **wide variety of databases**.

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**NumPy** : *simple example 1 – using the array function*



```
In [1]: from numpy import *

In [2]: a = array([[4., 3., 8.], [2., 1., 9.]])

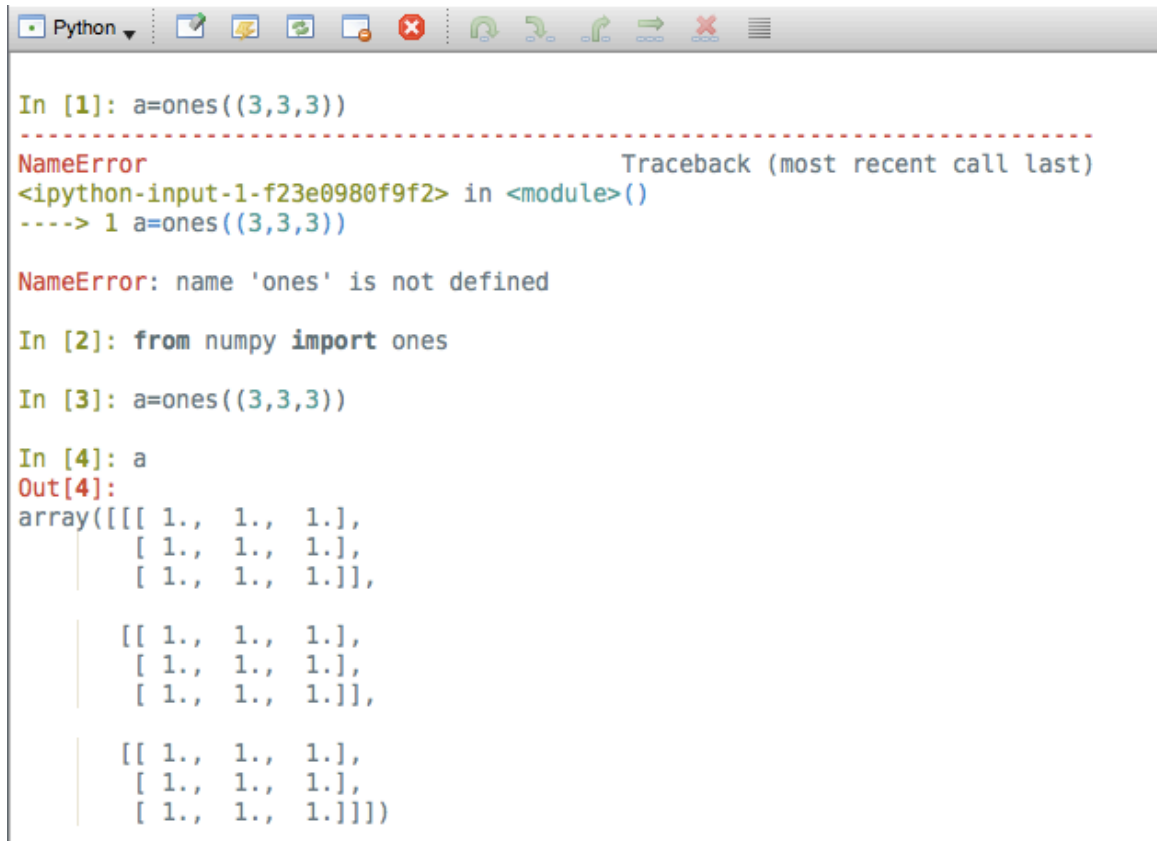
In [3]: a
Out[3]:
array([[ 4.,  3.,  8.],
        [ 2.,  1.,  9.]])

In [4]: a.shape
Out[4]: (2, 3)
```

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**NumPy** : *simple example 2 – using the zeros / ones functions*



```
Python
In [1]: a=ones((3,3,3))
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-f23e0980f9f2> in <module>()
----> 1 a=ones((3,3,3))

NameError: name 'ones' is not defined

In [2]: from numpy import ones

In [3]: a=ones((3,3,3))

In [4]: a
Out[4]:
array([[[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

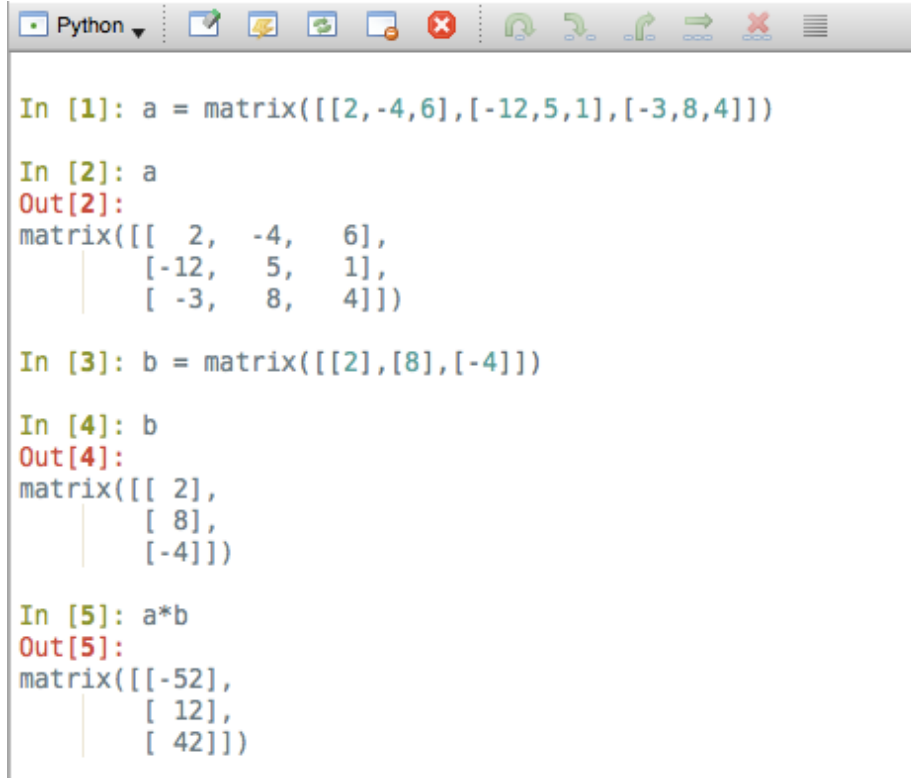
       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
```



# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**NumPy** : *simple example 3 – two-dimensional array – a matrix*



```
Python
In [1]: a = matrix([[2,-4,6],[-12,5,1],[-3,8,4]])

In [2]: a
Out[2]:
matrix([[ 2, -4,  6],
        [-12,  5,  1],
        [-3,  8,  4]])

In [3]: b = matrix([[2],[8],[-4]])

In [4]: b
Out[4]:
matrix([[ 2],
        [ 8],
        [-4]])

In [5]: a*b
Out[5]:
matrix([[ -52],
        [ 12],
        [ 42]])
```

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

## SciPy :

- This package is **open-source for mathematics, science, and engineering**
- Some of its most attractive key points are:
  - SciPy **depends on the NumPy** library and that provides convenient and fast N-dimensional array manipulation
  - SciPy **is built to work with NumPy arrays**, and provides many numerical routines like routines for numerical integration and optimization

## SciPy and NumPy :

- run on **all popular operating systems**
- are **quick to install** and are free of charge
- are **easy to use** and so **powerful** that some of the world's leading scientists and engineers depend on it
- easy to use when manipulating numbers, **display and publish the results**

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**SciPy :** *simple example 1 – working with polynomials*



```
Python
In [1]: from scipy import *
In [2]: x = poly1d([4,2,6])
In [3]: x
Out[3]: poly1d([4, 2, 6])
In [4]: print(x)
      2
4 x + 2 x + 6
In [5]: print(x*x)
      4      3      2
16 x + 16 x + 52 x + 24 x + 36
In [6]: print(x.integ(k=5))
      3      2
1.333 x + 1 x + 6 x + 5
In [7]: print(x.deriv())
      8 x + 2
In [8]: x([4,5])
Out[8]: array([ 78, 116])
```

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**SciPy :** *simple example 2 – using the FFT and IFFT*



```
Python
In [1]: from scipy.fftpack import fft, ifft
In [2]: from numpy import array
In [3]: a = array([2.3, 3.1, 4.2, -1.8, 1.6, 5.9])
In [4]: a
Out[4]: array([ 2.3,  3.1,  4.2, -1.8,  1.6,  5.9])
In [5]: b = fft(a)
In [6]: b
Out[6]: array([ 15.3+0.j          ,  5.7+0.17320508j, -6.9+4.67653718j,
                | 0.9+0.j          , -6.9-4.67653718j,  5.7-0.17320508j])
In [7]: b_inverse = ifft(b)
In [8]: b_inverse
Out[8]: array([ 2.3 +0.00000000e+00j,  3.1 -2.96059473e-16j,  4.2 +0.00000000e+00j,
                | -1.8 +9.17214811e-16j,  1.6 +0.00000000e+00j,  5.9 -6.21155338e-16j])
In [9]: a_sum = a.sum()
In [10]: a_sum
Out[10]: 15.300000000000002
```

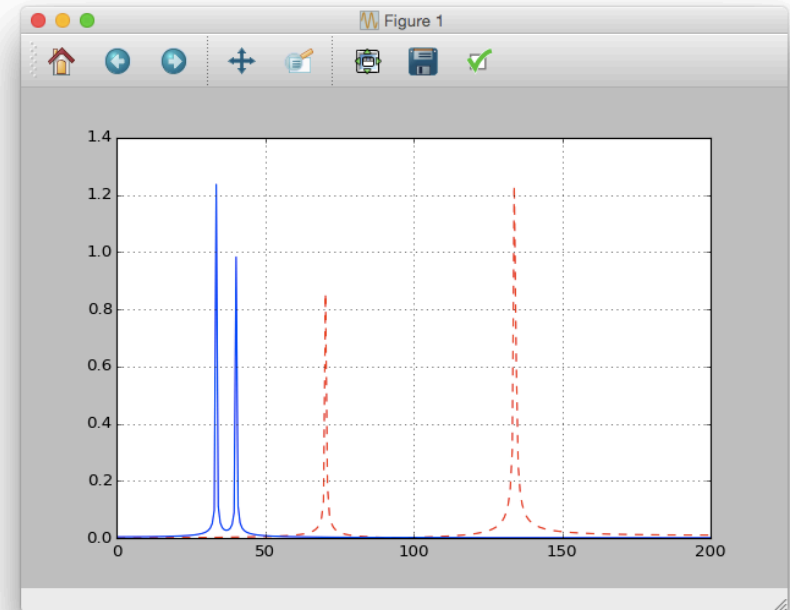
# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**SciPy :** *simple example 3 – plot the FFT of the sum of two series*

code:

```
from scipy.fftpack import fft
import numpy as np
import matplotlib.pyplot as plt
F = 600 # Sample frequency:
T = 1.0 / F # Period T:
a = np.linspace(0.0, F*T, F)
b = np.sin(100.0 * 4.0*np.pi*a) + 0.6*np.sin(70.0 * 3.0*np.pi*a)
c = np.cos(50.0 * 2.0*np.pi*a) + 0.8*np.sin(60.0 * 2.0*np.pi*a)
bf = fft(b); cf = fft(c)
af = np.linspace(0.0, 1.0/(3.0*T), F/2)
plt.plot(af, 3.0/F * np.abs(bf[0:F/2]),'r--')
plt.plot(af, 2.5/F * np.abs(cf[0:F/2]),'b-')
plt.grid(); plt.pause(1) # plt.show()
```



# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

## Matplotlib :

- The development of Matplotlib was initiated by John Hunter (1968-2012)
- It works great with NumPy and SciPy providing the platform for visualizing results
- This package is built for Python and provides a powerful set of 2D plotting functions
- It can be used in:
  - python scripts
  - the python and iPython shell
  - web application servers,
  - additional graphical user interface toolkits
- The user can easily generate plots, histograms, bar charts, scatterplots, and much more
- Users have the ability to manipulate the axis dimensions and labeling, title, legend, grid, fitting, zooming, etc.
- Working with Matplotlib resembles working with Matlab plotting
- Matplotlib is an open source package

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

## Matplotlib :

### – There are several toolkits for Matplotlib:

- General Toolkits
  - [mplot3d](#)
  - AxesGrid
  - MplDataCursor
  - GTK Tools
  - Excel Tools
  - Natgrid
- Mapping Toolkits
  - Basemap
  - Cartopy
- High-Level Plotting
  - seaborn
  - ggplot
  - prettyplotlib

(source: [http://matplotlib.org/mpl\\_toolkits/index.html](http://matplotlib.org/mpl_toolkits/index.html) )

# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**Matplotlib :**     *simple example 1 – using mplot3d and a wireframe*

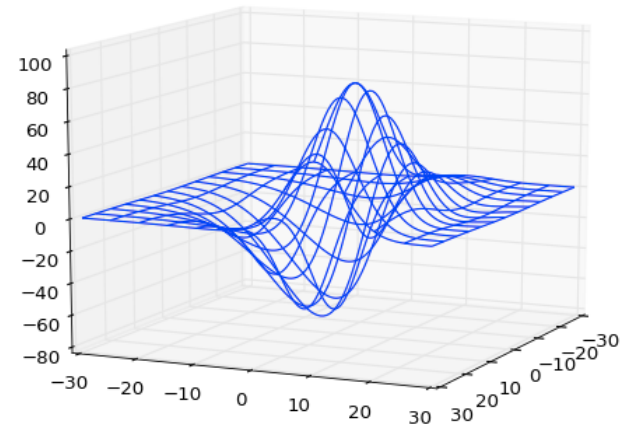
– There are several toolkits for Matplotlib:

- General Toolkits
  - `mplot3d` example:

code:

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.pause(1) # plt.show()
```



(source: [http://matplotlib.org/mpl\\_toolkits/index.html](http://matplotlib.org/mpl_toolkits/index.html) )



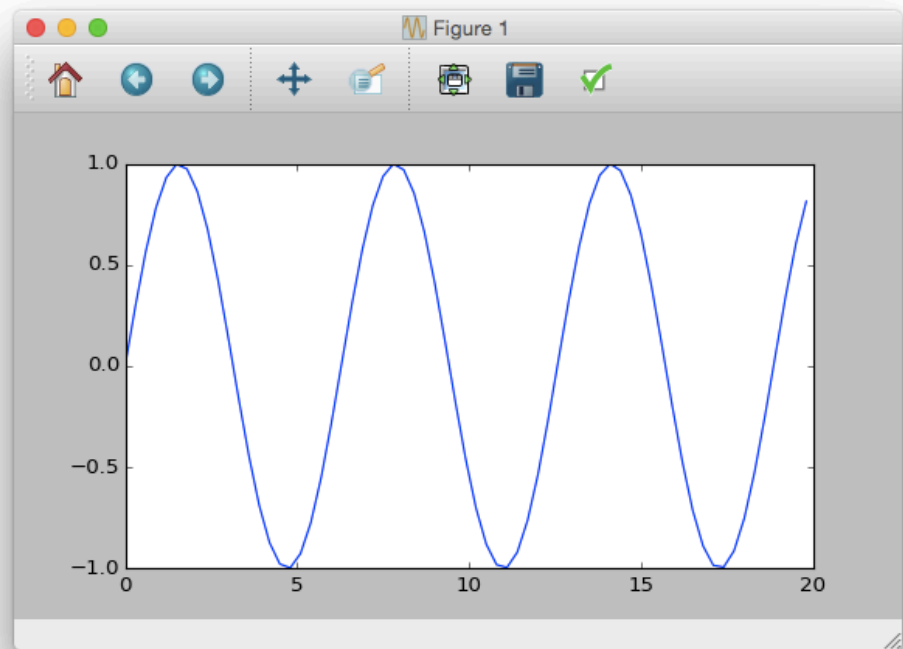
# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**Matplotlib :**     *simple example 2 – plotting a simple sinusoid*

code:

```
from numpy import *  
import matplotlib.pyplot as plt  
x = arange(0,20.,0.3)  
y = sin(x)  
ll = plt.plot(x,y)  
plt.pause(1) # plt.show()
```



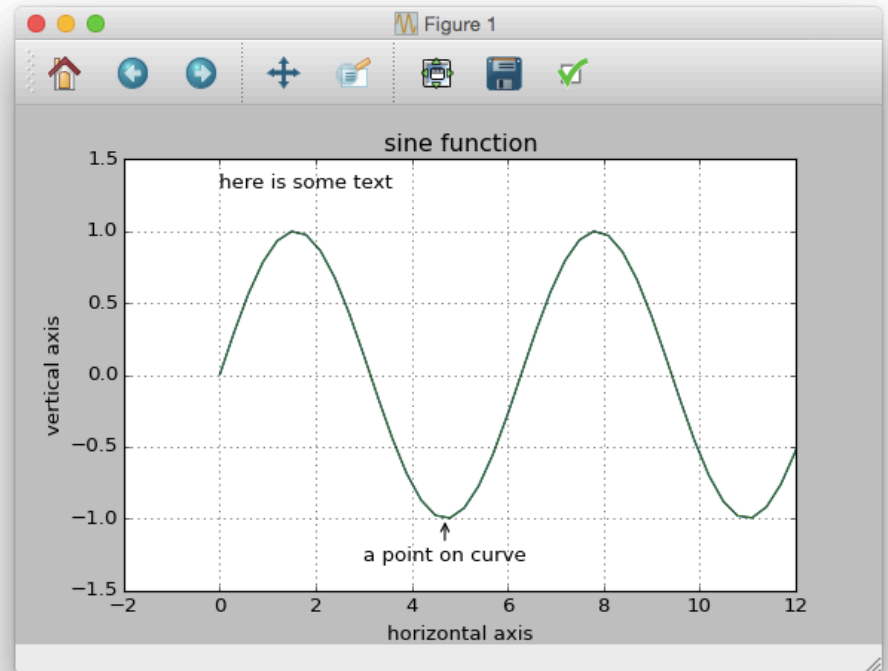
# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**Matplotlib :** *simple example 3 – plotting a simple sinusoid with annotation*

code:

```
ll = plt.plot(x,y)
xl = plt.xlabel('horizontal axis')
yl = plt.ylabel('vertical axis')
ttl = plt.title('sine function')
ax = plt.axis([-2, 12, -1.5, 1.5])
grd = plt.grid(True)
txt = plt.text(0,1.3,'here is some text')
ann = plt.annotate('a point on curve',xy=(4.7,-1),xytext=(3,-1.3), arrowprops=dict(arrowstyle='->'))
plt.pause(1) # plt.show()
```



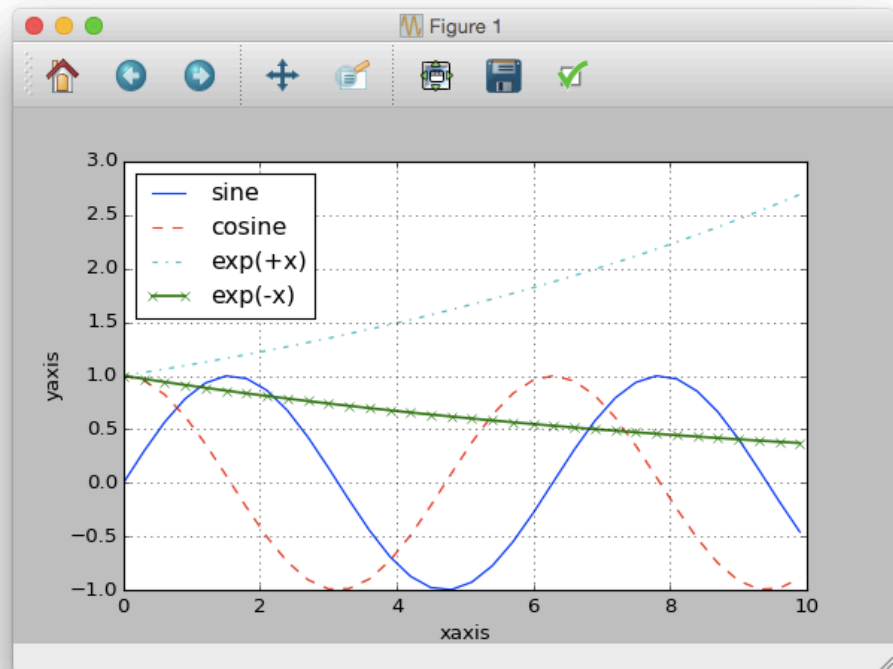
# Python for Data Analysis and Scientific Computing

- Python modules, packages and scientific packages

**Matplotlib :** *simple example 4 – plotting the sin and cos functions with annotation*

code:

```
x = arange(0.,10,0.3)
a = sin(x); b = cos(x);
c = exp(x/10); d = exp(-x/10)
plt.plot(x,a,'b-',label='sine')
plt.plot(x,b,'r--',label='cosine')
plt.plot(x,c,'c-.',label='exp(+x)')
plt.plot(x,d,'gx-',linewidth = 1.5,label='exp(-x)')
plt.legend(loc='upper left')
plt.grid()
plt.xlabel('xaxis')
plt.ylabel('yaxis')
plt.pause(1) # plt.show()
```



# Language specifics

- Basic arithmetic operations

- The basic arithmetic operations are 5:
  - Addition (+)
    - $a = 3 + 4$  -> creates an integer
    - $b = 6 + 7.$  -> creates a floating point number (higher precision)
  - Subtraction (-)
    - $c = 8 - 12.$  -> creates a floating point number
  - Multiplication (\*)
    - $d = 2 * 4$  -> creates an integer
  - Division (/)
    - $e = 6 / 3.$  -> creates a floating point number
  - Modulo (%)
    - $f = 4 \% 3$  -> will return '1'
- The other widely used are:
  - Power
    - $g = 2 ** 3$  -> will produce '8'
  - Square root
    - $h = 8 **.5$  -> will produce '2.8284271247461903' ( hint: you can also use 'math.sqrt(8)' )
  - Explicit integer rounding
    - $i = 5 // 3.$  -> will produce '1.0'

# Language specifics

- Assignment operators, data types, containers
  - Assigning a **scalar**, **vector** or a **matrix** to **variables** is used to bind **values to names** and to modify attributes or items of different objects
  - A simple assignment works as follows:

```
In [1]: x = 5
```

- the expression on the **right hand side is evaluated**
  - the corresponding **object is created** and stored in memory
  - a **name on the left hand side is assigned**, or bound, to the right hand side object
- Once assigned, the object's content can be replaced easily like this:

```
In [2]: x = [4,6,3,5,4]
```

**This is because variables in Python are references to objects that can be changed**

# Language specifics

- Assignment operators, data types, containers

- We can easily make a copy of one variable to another:

```
In [3]: y = x
```

- ... then try if they are equal:

```
In [4]: x is y
```

```
Out[4]: true
```

- Each element in a list can be changed easily:

```
In [5]: y[2] = 'Python rocks'
```

```
In [6]: y
```

```
Out[6]: [4, 6, 'Python rocks', 5, 4]
```

Note: notice that enumeration of elements in a object start from '0' not from '1'

# Language specifics

- Assignment operators, data types, containers
  - Mutable vs. Immutable objects in Python:
    - **Mutable** objects can **change** their value
    - **Immutable** objects **do not change** their assigned value

**Note:** Immutable objects can be tricky as some immutable objects may look like they change their values, but they actually are not (such are tuples, strings, integers, etc.)

- To check the value of an object we use the ***'id()'*** function

Example:

```
In [7]: a = 128
```

```
In [8]: a
```

```
Out[8]: 128
```

# Language specifics

- Assignment operators, data types, containers
  - To check the value of an object we use the *'id()'* function

Example:

```
In [9]: type(a)
Out[9]: int
In [10]: id(a)
Out[10]: 4297335296
```

- **'int'** are **immutable** objects, therefore their **values can not be changed**, but ...

```
In [11]: a = 256
In [12]: a
Out[12]: 256    ... Note: your id() value will be different
```



# Language specifics

- Assignment operators, data types, containers

- A variable can refer to a different object (in our case integer) because:

variables in Python are references to objects

- Let's test that statement:

```
In [13]: id(a)
```

```
Out[13]: 4297339392 ... Note: this value is different than what I had before
```

- ... let's see what would happen if I changed the value of 'a' back to '128':

```
In [14]: a = 128
```

```
In [15]: a
```

```
Out[15]: 128
```

```
In [16]: id(a)
```

```
Out[16]: 4297335296 ... Note: this is the same value as before the change
```

- We didn't change the immutable value of 'a', rather **we changed its reference** to a different object

# Language specifics

- Assignment operators, data types, containers
  - Multiple assignments are possible in Python:
    - 1) `a = b = c = 50`
    - 2) `a, b, c = 'Alex', 12, 24`
  - In cases when changing a value is needed, a mutable object should be used instead
  - **Immutable** objects are the most common data types in Python like:
    - **strings** - sequence of values (there is no *'char'* type in Python)
    - **tuples** - sequences just like 'list's, but are immutable and use parenthesis
    - **bytes** - each byte is an 8-bit object represented by integers in the range  $0 \leq x < 256$  ... ( $2^8$ )
    - ... more on that later
  - **Mutable** objects are:
    - **lists** - can change (unlike tuples) and use square brackets
    - **byte arrays** - this object is a mutable array created by the `'bytearray()'` constructor
    - ... more on that later

# Language specifics

- Assignment operators, data types, containers
  - **None** – is used to signify the absence of a value in different situations. It is returned from functions that don't return anything
  - **NotImplemented** – functions may return this value to show that a **comparison** or an operation **is not implemented with respect to another type** in the expression
  - **Numbers** – created by numeric literals that once assigned do not change their value hence they are immutable
    - **integers** – can be positive and negative
      - **int** – represent numbers in an unlimited range and depend on memory only
      - **bool** – take two values 'True' = 1 or 'False' = 0
    - **float** – they represent double precision floating point numbers
    - **complex** – numbers as a **pair of machine-level double precision floating point** numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number 'a' can be retrieved through:
      - **a.real** – represents the real part of the double precision complex floating point number
      - **a.imag** – represents the imaginary part of the double precision complex floating point number
  - **Ellipsis** – is an object that can appear in slice notation. It may be used to indicate a placeholder for the rest of an array dimensions **not explicitly specified**.

# Language specifics

- Assignment operators, data types, containers

There are **six built-in constants** used in Python: *none*, *NotImplemented*, *True*, *False*, *Ellipsis*, *\_\_debug\_\_*

- **None**

- is special because it holds **no value**
- is a **constant** that lives in the built-in namespace of Python
- it is **returned** usually **by methods** or collections (ex: dictionaries, etc.)
- it **can not be called** by methods such as len() ... if called a 'TypeError' will occur

Example:

```
In [16]: z = None
```

```
In [17]: len(z)
```

```
-----  
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-64-d46262532051> in <module>()  
----> 1 len(z)
```

```
TypeError: object of type 'NoneType' has no len()
```

# Language specifics

- Assignment operators, data types, containers
  - None
    - None is **not the same** thing as **empty**
    - Example: a list can be empty and will have a length of 0
    - It is strongly recommended to **use an 'if' statement to check for None** values
    - ... more on 'if' statements later

Example:

```
def check(c):  
    # Here we check for None value. If it is, then print 'Value is None'  
    if c == None:  
        print('Value is None')  
    else:  
        print(len(c))
```

*The output is:*

```
In [18]: check('Olaaaaaa')
```

```
8
```

```
In [19]: check(None)
```

```
Value is None
```

# Language specifics

- Assignment operators, data types, containers
  - None
    - None is often **tested when accessing elements in a dictionary** by using the 'get' method
    - None is a **special value to the dictionary** that means “not found”
    - None means that the object is not present and **the variable doesn't point to an object**

Example using Dictionary:

```
def dictionary(c):  
    objects = {"bar" : 1, "car" : 2, "office" : 3, "police" : 4}  
    v = objects.get(c)  
    if v == 2:  
        print('Maserati')  
    else:  
        print('Not existing')
```

*The output is:*

```
In [20]: dictionary('car')  
Maserati  
In [21]: dictionary('truck')  
Not existing
```

... more on Dictionary later

# Language specifics

- Assignment operators, data types, containers
  - **NotImplemented**
    - can be reassigned to hold another value, which is called **shadowing**

**Warning: this action will change its meaning and it does not evoke a `SyntaxError` and should not be attempted**

- this alone makes **NOT a true constant** (because constant shouldn't change)

Example:

```
In [18]: None = 'Python is good'
... SyntaxError: can't assign to keyword
In [19]: NotImplemented
Out[19]: NotImplemented
In [20]: NotImplemented = 'Python is good'
In [21]: NotImplemented
Out[21]: 'Python is good'
```

... more on that later

# Language specifics

- Assignment operators, data types, containers

- True / False:

- are *bool* type constants used in Python
    - no assignments can be done to them
    - in case such attempt is made a `SyntaxError` will occur

```
In [22]: type(True)
```

```
Out[22]: bool
```

- Ellipsis:

- a special value used mostly with slicing syntax for user-defined container types
    - it holds a single value
    - a single object is associated with this value
    - the object is accessed through the literal `...` or the built-in name *Ellipsis*

- `__debug__` :

- it is a special type of Boolean (... although the only true Booleans are: `True` and `False`)
    - this constant is true if Python was not started with a basic optimization option
    - used as a convenient way to insert debugging assertions into your code

```
if __debug__:
```

```
    if not expression: raise AssertionError
```



# Language specifics

- Assignment operators, data types, containers

- Integers:

- They can be positive and negative
    - They are **two types** of integers: **Integers** *'int'* and **Boolean** *'bool'*

- Integers ([int](#)):

- » represent numbers in an unlimited range and depend on memory only

- In [23]: x=5

- In [24]: type(x)

- Out[24]: int

- Booleans ([bool](#)):

- » are a subclass of *'int'*

- » they take two values 'True' = 1 or 'False' = 0

- » they are NOT the only Boolean objects (despite of what some literature says)

- if *condition\_met*:

- var = True

- if *var*: ... execute some code here

# Language specifics

- Assignment operators, data types, containers
  - Constants:
    - To sum it up, here is what calling the ***‘type’*** each of the six constants will return:

```
In [25]: type(None)  
Out[25]: NoneType
```

```
In [26]: type(NotImplemented)  
Out[26]: NotImplementedType
```

```
In [27]: type(Ellipsis)  
Out[27]: ellipsis
```

```
In [28]: type(True)  
Out[28]: bool
```

```
In [29]: type(False)  
Out[29]: bool
```

```
In [30]: type(__debug__)  
Out[30]: bool
```

# Language specifics

- Assignment operators, data types, containers
  - Floating point – float:
    - the floating point type represents **double precision floating point** numbers
    - double precision means 8 bytes (8-bits each) or **64-bits** (rather than 32-bits for single precision)
    - the underlying machine architecture (ex: written in C) is **handling any possible overflow** and **errors may occur** (truncation or rounding)
    - **overflow is unwanted** event since numbers, larger than what the program can handle, occurred
    - in the case of overflow the program must be **re-written to avoid** such large values
    - **single-precision floating point numbers are not supported in Python**
  - pro
    - the **error** in the final result **is much smaller** since the precision is much higher when using floats
  - con
    - **processor and memory usage is much larger** when dealing with floating point numbers

In [31]: x=3.45

In [32]: type(x)

Out[32]: float

# Language specifics

- Assignment operators, data types, containers
  - Complex:
    - complex number is any number that has **two values** and looks like this: ***real + imag\*1j***
    - these numbers represent complex structure of a **pair of machine-level double precision floating point numbers**
    - this is **similar for floating point** numbers as well
    - the **real** and **imaginary** parts of a **complex** number 'Z' can be viewed by using '**real**' and '**imag**' in the following way: Z.real and Z.imag
    - both attributes '**real**' and '**imag**' are read-only

```
In [33]: x=3+4j
```

```
In [34]: x
```

```
Out[34]: (3+4j)
```

```
In [35]: type(x)
```

```
Out[35]: complex
```

```
In [36]: x.real
```

```
Out[36]: 3.0
```

```
In [37]: x.imag
```

```
Out[37]: 4.0
```

# Language specifics

- Assignment operators, data types, containers (*aka collections or sequences*)
  - **Sequences** – represent **finite sets indexed by a non-negative number** in the range of 0, 1, ...,  $n-1$ . Accessing each individual item in a sequence 'a' is done like this:  $a[m]$ , where  $0 < m < n-1$ .

Using the built-in function '**len()**' can reveal the number of items in a sequence

Sequences differ based on their mutability:

- **Immutable sequences** – these cannot change once they are created:
  - **strings** – a **sequence** of values
  - **tuples** – items of a tuple can be **arbitrary** objects. Tuples of two or more items are formed by comma-separated lists
  - **bytes** – the bytes object is an **array**. Bytes are 8-bits integers in the range between  $0 \leq x < 256$  ( $2^8$ )
  - **int** – are implemented using long in C, which gives them at least 32 bits of precision
  - **bool** – is another primitive type and is a subtype of **int**. **True** and **False** are singletons and cannot be modified
- **Mutable sequences** – these can be changed after they are created:
  - **lists** – items in a list are **arbitrary** objects formed by placing a **comma-separated list of expressions** in square brackets
  - **dictionaries** – these are also **arbitrary** objects with keys that have corresponding values
  - **byte arrays** – the byte array object is a mutable **array**, that has the same functionality as the immutable bytes object

# Language specifics

- Assignment operators, data types, containers
  - Strings:
    - a sequence of values that represent Unicode code points (range [U+0000 - U+10FFFF] )
    - Python doesn't have a 'char' type
    - every code point in a string is represented as a string object with length 1
    - some useful built-in functions:
      - `ord()` converts a **one-character** string to its integer Unicode code point representation
      - `int()` converts a **multiple-character** string to an integer (*only if string of numbers*)
      - `chr()` converts an integer to a string in the range [0 - 10FFFF]
      - `str.encode()` can be used to convert a str to bytes using the given text encoding
      - `bytes.decode()` can be used to achieve the opposite

Example:

```
In [38]: x='wer'
In [39]: type(x)
Out[39]: str
In [40]: x[1]
Out[40]: 'e'
In [41]: ord(x[1])
Out[41]: 101
In [42]: chr(ord(x[1]))
Out[42]: 'e'
```

# Language specifics

- Assignment operators, data types, containers
  - **Tuples**: ... *what are they?*
    - they are **sequences** of **immutable** Python objects
    - items of a tuple can be **arbitrary** objects
    - tuples use **parenthesis** to be created
    - they are just **like lists**, but the latter use square brackets
    - an **empty tuple** can be formed by an **empty pair of parentheses**
    - a tuple of one item is called a '**singleton**'
    - tuples of two or more items are formed by **comma-separated lists**

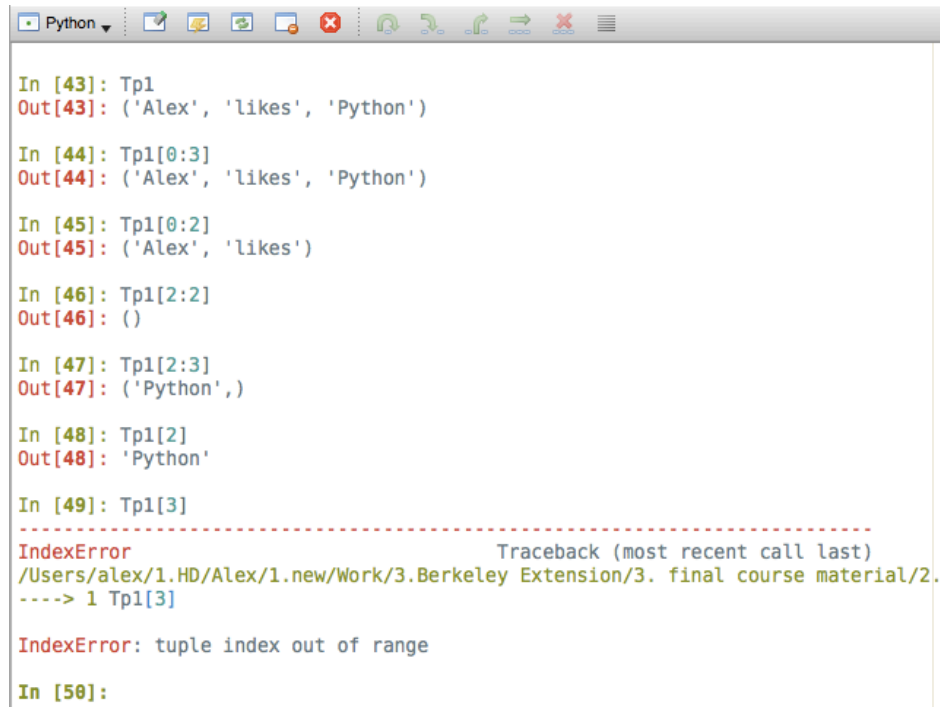
Examples:

<code>Tp1 = "Alex", "likes", "Python"</code>	<code>#</code>	tuple with strings only
<code>Tp2 = ('cars', 2, 'bars', 3, 'born', 2000)</code>	<code>#</code>	tuple with strings and numbers mixed
<code>Tp3 = (23, 45, 31, 49, 52, 64)</code>	<code>#</code>	tuple with numbers only
<code>Tp4 = (23, )</code>	<code>#</code>	a singleton (... notice the comma!)
<code>Tp5 = ()</code>	<code>#</code>	an empty tuple

# Language specifics

- Assignment operators, data types, containers
  - **Tuples**: ... *accessing*
    - accessing different elements in tuples is **done by using square brackets** and the index or sequence of indices to obtain a particular value or series of values

Example:



```
Python
In [43]: Tp1
Out[43]: ('Alex', 'likes', 'Python')

In [44]: Tp1[0:3]
Out[44]: ('Alex', 'likes', 'Python')

In [45]: Tp1[0:2]
Out[45]: ('Alex', 'likes')

In [46]: Tp1[2:2]
Out[46]: ()

In [47]: Tp1[2:3]
Out[47]: ('Python',)

In [48]: Tp1[2]
Out[48]: 'Python'

In [49]: Tp1[3]
-----
IndexError                                Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
----> 1 Tp1[3]

IndexError: tuple index out of range

In [50]:
```

... class exercise



# Language specifics

- Assignment operators, data types, containers
  - **Tuples**: ... *updating*
    - since tuples are immutable objects they **cannot be updated** so their elements are set permanently
    - **rather**, portions of existing tuples can be taken to **create new tuples**

Example:



```
In [50]: Tp1
Out[50]: ('Alex', 'likes', 'Python')

In [51]: Tp2
Out[51]: ('cars', 2, 'bars', 3, 'born', 2000)

In [52]: Tp6 = Tp1 + Tp2

In [53]: Tp6
Out[53]: ('Alex', 'likes', 'Python', 'cars', 2, 'bars', 3, 'born', 2000)

In [54]: Tp7 = (Tp1[0:2], Tp2[2])

In [55]: Tp7
Out[55]: (('Alex', 'likes'), 'bars')

In [56]: Tp7[0]
Out[56]: ('Alex', 'likes')

In [57]: Tp7[1]
Out[57]: 'bars'

In [58]:
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - Tuples: ... *basic operations*
    - **length** – using the command '**len()**' displays the number of members in a tuple
    - **membership** – checks if an element exist in a tuple
    - **concatenation** – using a simple '+' sign. Works for numbers and strings
    - **repetition** – using simple '\*' sign. Simply repeats a string number of times
    - **iteration** – goes through a loop and displays each member in a tuple

Example:

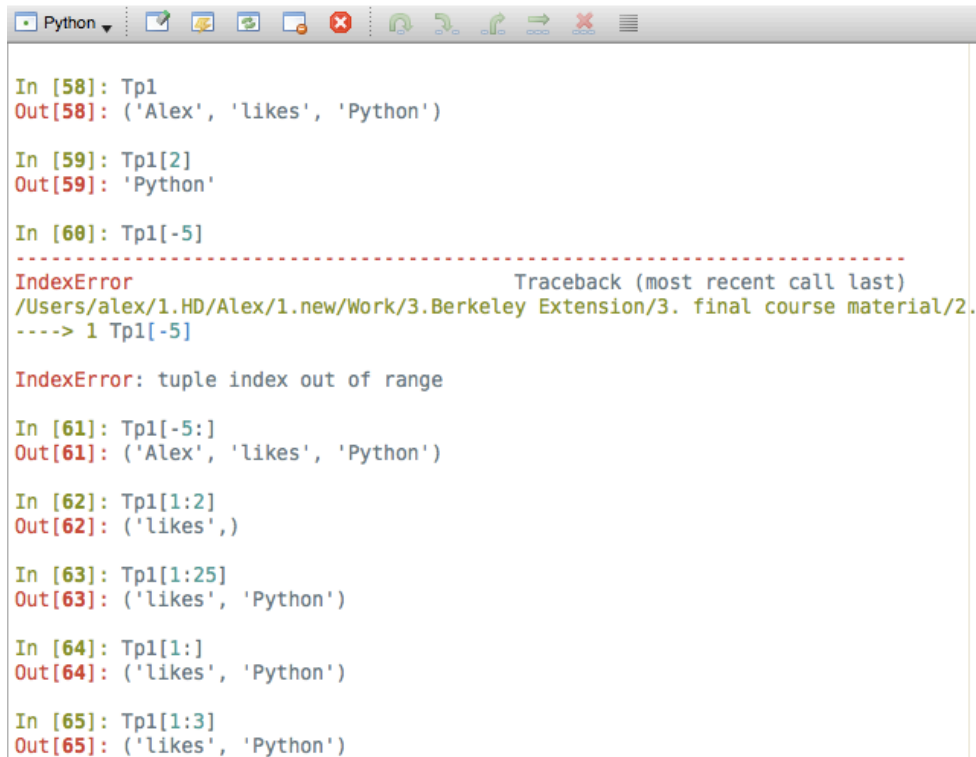
Tuple Operations		
Description	Expression	Result
Length	len((41, 12, 34, 52))	4
Membership	41 in (41, 12, 34, 52)	TRUE
Concatenation	('Python')+(' is cool')	'Python is cool'
Repetition	('Python') * 2	('PythonPython')
Iteration	for x in (41, 12, 34, 52): print x,	41 12 34 52

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - Tuples: ... *slicing*
    - parts of tuples can be accessed
    - this is similar for strings

Example:



```
Python
In [58]: Tp1
Out[58]: ('Alex', 'likes', 'Python')

In [59]: Tp1[2]
Out[59]: 'Python'

In [60]: Tp1[-5]
-----
IndexError                                Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
----> 1 Tp1[-5]

IndexError: tuple index out of range

In [61]: Tp1[-5:]
Out[61]: ('Alex', 'likes', 'Python')

In [62]: Tp1[1:2]
Out[62]: ('likes',)

In [63]: Tp1[1:25]
Out[63]: ('likes', 'Python')

In [64]: Tp1[1:]
Out[64]: ('likes', 'Python')

In [65]: Tp1[1:3]
Out[65]: ('likes', 'Python')
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - **Tuples**: ... *tuple functions*
    - `min()` / `max()`:
      - shows the **min/max value** in a tuple of several elements
      - works for tuples with numbers only or strings only. It **does not work for mixed tuples**

Example:

```
Python
In [66]: Tp1
Out[66]: ('Alex', 'likes', 'Python')

In [67]: min(Tp1)
Out[67]: 'Alex'

In [68]: max(Tp1)
Out[68]: 'likes'

In [69]: Tp2
Out[69]: ('cars', 2, 'bars', 3, 'born', 2000)

In [70]: min(Tp2)
-----
TypeError                                Traceback (most recent call last)
<ipython-input-70-d6f544a0378e> in <module>()
----> 1 min(Tp2)

TypeError: unorderable types: int() < str()

In [71]: Tp3
Out[71]: (23, 45, 31, 49, 52, 64)

In [72]: min(Tp3)
Out[72]: 23

In [73]: max(Tp3)
Out[73]: 64
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - **Tuples**: ... *tuple functions*
    - `len()`:
      - shows the length/number of elements in a tuple. We *already saw examples using 'len()'*
    - `tuple()`:
      - converts lists in tuples

Example:

```
Python
In [74]: Tp8=[Tp1[0:1],Tp2[1]]

In [75]: Tp8
Out[75]: [('Alex',), 2]

In [76]: whos
Variable Type Data/Info
-----
Tp1      tuple  n=3
Tp2      tuple  n=6
Tp8      list   n=2
name     str    __loader__

In [77]: Tp8 = tuple(Tp8)

In [78]: Tp8
Out[78]: (('Alex',), 2)

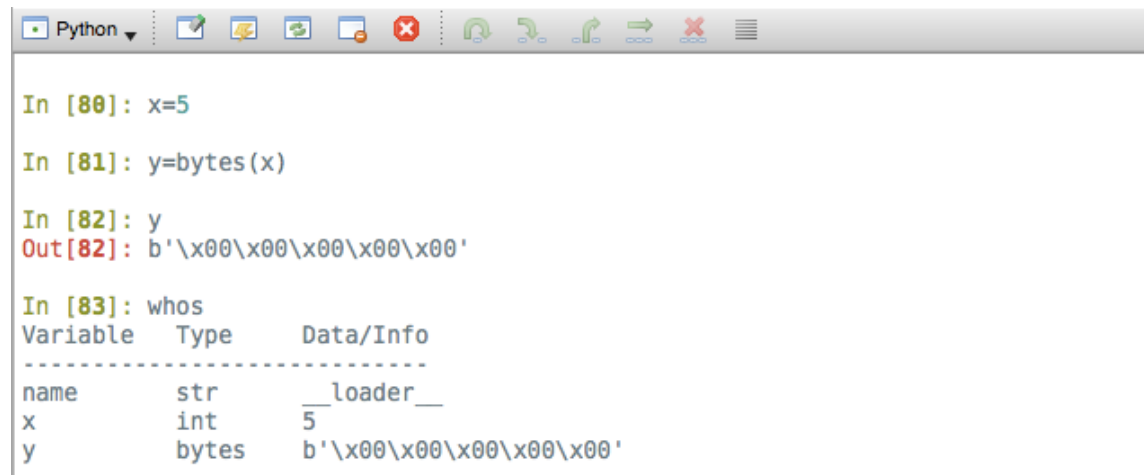
In [79]: whos
Variable Type Data/Info
-----
Tp1      tuple  n=3
Tp2      tuple  n=6
Tp8      tuple  n=2
name     str    __loader__
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - Bytes:
    - the bytes object is an **array**
    - Bytes are **8-bits integers** in the range between  $0 \leq x < 256$  ( $2^8$ )
    - Bytes objects are **immutable**
    - the **built-in** function `bytes()` can be used to construct bytes objects

Example:



```
In [80]: x=5

In [81]: y=bytes(x)

In [82]: y
Out[82]: b'\x00\x00\x00\x00\x00'
```

```
In [83]: whos
Variable Type      Data/Info
-----
name     str          __loader__
x        int          5
y        bytes       b'\x00\x00\x00\x00\x00'
```

# Language specifics

- Assignment operators, data types, containers
  - **Lists:** ... *what are they?*
    - they are one of the two **most commonly used sequences** in Python (the other is Tuples)
    - it is a **sequence of immutable** Python objects, but Lists themselves are **mutable** (see example)
    - items in a list are different objects divided by **comma-separated list** of expressions
    - just like in strings, the first index in lists is '0'
    - list use **square brackets** to be created
    - they are just **like tuples**, but the latter use parenthesis
    - an empty list can be formed by an **empty pair of square brackets**

Examples:

Ls1= ["Alex", "likes", "Python"]	#	a list with strings only
Ls2 = ['cars', 2, 'bars', 3, 'born', 2000]	#	a list with strings and numbers mixed
Ls3 = [23, 45, 31, 49, 52, 64]	#	a list with numbers only
Ls4 = [23, ]	#	a list with one element
Ls5 = []	#	an empty list

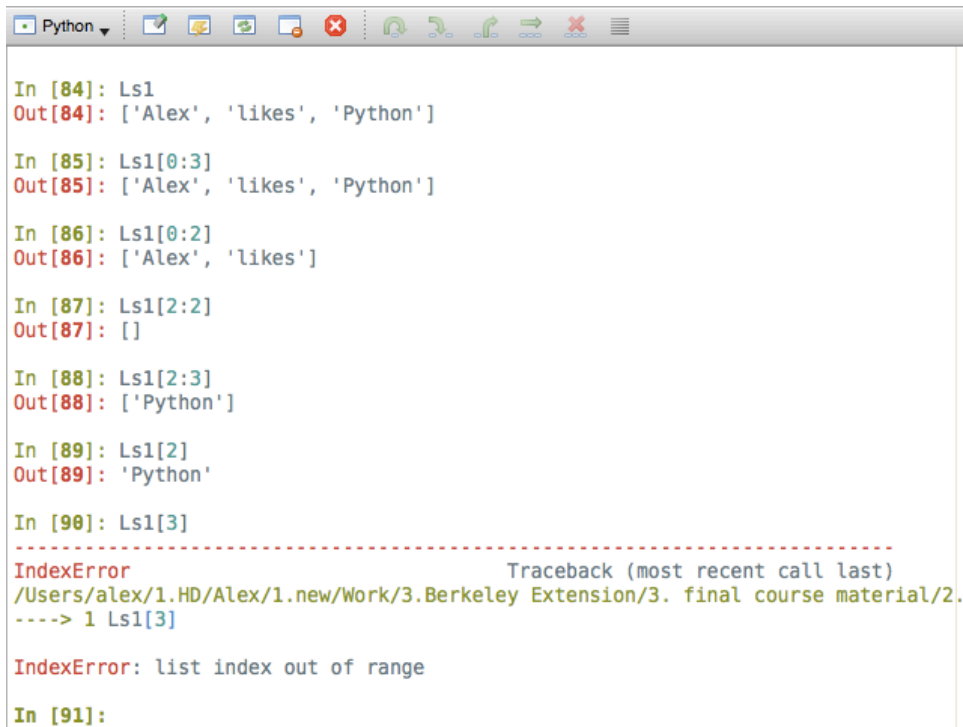
# Language specifics

- Assignment operators, data types, containers

- **Lists:** ... *accessing*

- just like in Tuples, accessing different elements in lists is done by using **square brackets**
    - the index or sequence of indices to obtain a particular value or series of values is the same

Example:



```
Python
In [84]: Ls1
Out[84]: ['Alex', 'likes', 'Python']

In [85]: Ls1[0:3]
Out[85]: ['Alex', 'likes', 'Python']

In [86]: Ls1[0:2]
Out[86]: ['Alex', 'likes']

In [87]: Ls1[2:2]
Out[87]: []

In [88]: Ls1[2:3]
Out[88]: ['Python']

In [89]: Ls1[2]
Out[89]: 'Python'

In [90]: Ls1[3]
-----
IndexError                                Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
----> 1 Ls1[3]

IndexError: list index out of range

In [91]:
```

... class exercise



# Language specifics

- Assignment operators, data types, containers
  - Lists: ... *updating*
    - unlike tuples, lists **can be updated** by accessing each individual member or a group of members
    - **adding** at the end of a list can be done by using the '**append()**' method
    - **deleting** a list or an individual member of a list is done by using the '**del()**' function

Example:

```
Python
In [91]: Ls1
Out[91]: ['Alex', 'likes', 'Python']

In [92]: del(Ls1[2])

In [93]: Ls1
Out[93]: ['Alex', 'likes']

In [94]: Ls1.append('bars')

In [95]: Ls1
Out[95]: ['Alex', 'likes', 'bars']

In [96]: Ls1.append(Ls2[0])

In [97]: Ls1
Out[97]: ['Alex', 'likes', 'bars', 'cars']

In [98]: Ls1[0]='John'

In [99]: Ls1
Out[99]: ['John', 'likes', 'bars', 'cars']
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - Lists: ... *basic operations*
    - **length** – using the command '**len()**' displays the number of members in a list
    - **membership** – checks if an element exist in a list
    - **concatenation** – using a simple '+' sign. Works for numbers and strings
    - **repetition** – using simple '\*' sign. Simply repeats a string number of times
    - **iteration** – goes through a loop and displays each member in a list

Example:

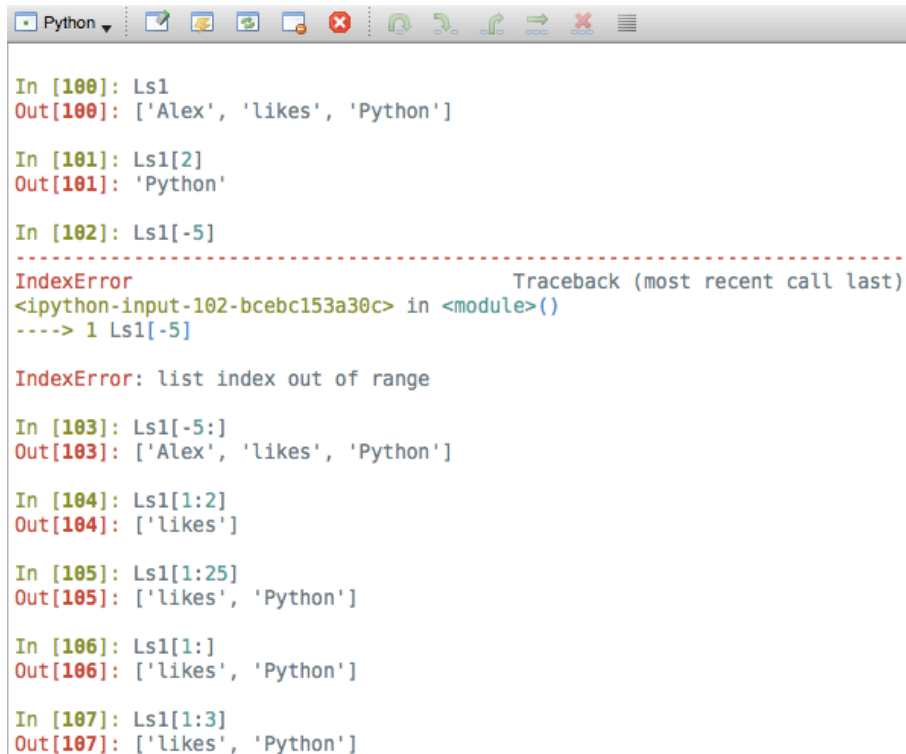
List Operations		
Description	Expression	Result
Length	len([41, 12, 34, 52])	4
Membership	41 in [41, 12, 34, 52]	TRUE
Concatenation	['Python']+[' is cool']	['Python', ' is cool']
Repetition	['Python'] * 2	['Python', 'Python']
Iteration	for x in [41, 12, 34, 52]: print x,	41 12 34 52

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - Lists: ... *slicing*
    - parts of lists can be accessed
    - this is similar for strings

Example:



```
Python
In [100]: Ls1
Out[100]: ['Alex', 'likes', 'Python']

In [101]: Ls1[2]
Out[101]: 'Python'

In [102]: Ls1[-5]
-----
IndexError                                Traceback (most recent call last)
<ipython-input-102-bcebc153a30c> in <module>()
----> 1 Ls1[-5]

IndexError: list index out of range

In [103]: Ls1[-5:]
Out[103]: ['Alex', 'likes', 'Python']

In [104]: Ls1[1:2]
Out[104]: ['likes']

In [105]: Ls1[1:25]
Out[105]: ['likes', 'Python']

In [106]: Ls1[1:]
Out[106]: ['likes', 'Python']

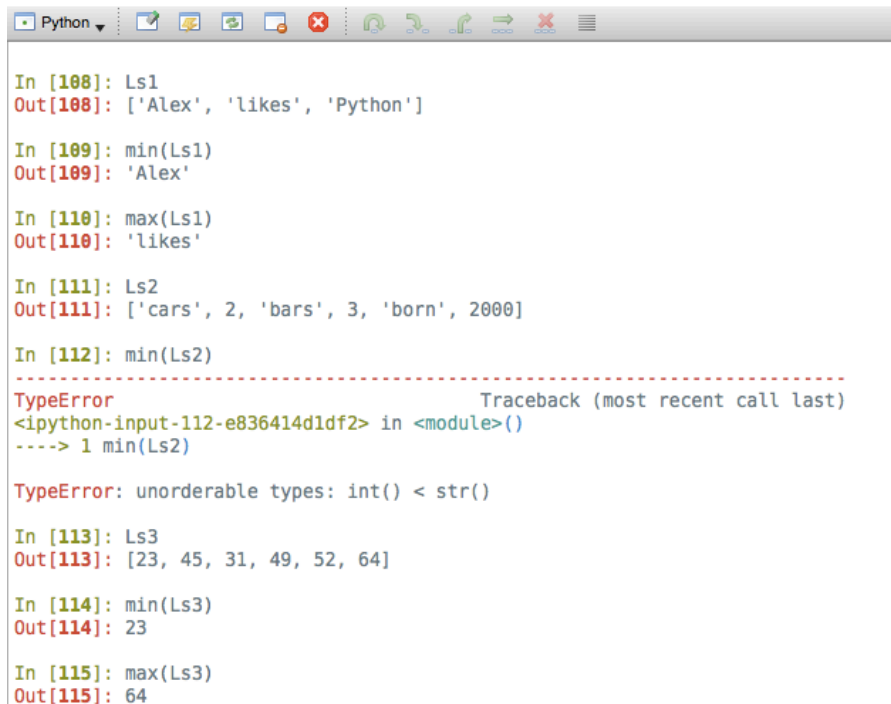
In [107]: Ls1[1:3]
Out[107]: ['likes', 'Python']
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - **Lists**: ... *list functions*
    - `min()` / `max()`:
      - shows the **min/max** value in a list of several elements
      - works for lists with numbers only or strings only. It **does not work for mixed lists**

Example:



```
In [108]: Ls1
Out[108]: ['Alex', 'likes', 'Python']

In [109]: min(Ls1)
Out[109]: 'Alex'

In [110]: max(Ls1)
Out[110]: 'likes'

In [111]: Ls2
Out[111]: ['cars', 2, 'bars', 3, 'born', 2000]

In [112]: min(Ls2)
-----
TypeError                                Traceback (most recent call last)
<ipython-input-112-e836414d1df2> in <module>()
----> 1 min(Ls2)

TypeError: unorderable types: int() < str()

In [113]: Ls3
Out[113]: [23, 45, 31, 49, 52, 64]

In [114]: min(Ls3)
Out[114]: 23

In [115]: max(Ls3)
Out[115]: 64
```

... class exercise

# Language specifics

- Assignment operators, data types, containers
  - **Lists**: ... *list functions*
    - `len()`:
      - shows the **length/number** of elements in a list. We *already saw examples using 'len()'*
    - `list()`:
      - converts **tuples to lists**

Example:

```
Python
In [116]: Ls8 = (Ls1[0:1], Ls2[1])
In [117]: Ls8
Out[117]: (['Alex'], 2)

In [118]: whos
Variable Type      Data/Info
-----
Ls1      list      n=3
Ls2      list      n=6
Ls8      tuple     n=2
name     str       __loader__

In [119]: Ls8 = list(Ls8)
In [120]: Ls8
Out[120]: [['Alex'], 2]

In [121]: whos
Variable Type      Data/Info
-----
Ls1      list      n=3
Ls2      list      n=6
Ls8      list      n=2
name     str       __loader__
```

... class exercise

# Language specifics

- Assignment operators, data types, containers

- **Lists:** ... *list functions*

- Methods:

- `list.index(obj)`      #      returns the lowest index of a particular existing object
      - `list.count(obj)`      #      checks the occurrence of particular object in a list
      - `list.append(obj)`      #      appends one object at the end of a list
      - `list.pop(-3)`      #      removes and returns the element '-3' from the list
      - `list.extend(seq)`      #      appends a sequence to a list
      - `list.sort([funct])`      #      sorts all objects in a given list with given 'funct'
      - `list.insert(ind,obj)`      #      inserts a particular object to a list with an offset index
      - `list.remove(obj)`      #      removes a particular object from a list
      - `list.reverse()`      #      reverse the order of objects in a list

# Language specifics

- Assignment operators, data types, containers
  - Dictionaries: ... *what are they?*
    - they consist of two **paired** fields: **keys** and **values**
    - possible **key appears just once** in the collection
    - they work like **associative arrays** or hashes
    - a dictionary **key** can be almost **any type**
    - **values** can be any **arbitrary** object
    - dictionaries are **formed** using **curly braces** (**{ }**)
    - values can be assigned/**accessed** with **square braces** (**[]**)
    - once set, particular keys can be **updated using** (**{ }**)

# Language specifics

- Assignment operators, data types, containers

- Dictionaries:

Example:

```
Python
In [122]: dictionary_1 = {'name': 'Sam', 'sex': 'male', 'age': 35}
In [123]: dictionary_1
Out[123]: {'name': 'Sam', 'age': 35, 'sex': 'male'}

In [124]: dictionary_2 = {} # creates an empty dictionary
In [125]: dictionary_2
Out[125]: {}

In [126]: dictionary_2['Sam'] = "Sam drinks beer"
In [127]: dictionary_2['age'] = "35 years old"

In [128]: dictionary_2
Out[128]: {'Sam': 'Sam drinks beer', 'age': '35 years old'}

In [129]: print(dictionary_2.keys())
dict_keys(['Sam', 'age'])

In [130]: print(dictionary_2.values())
dict_values(['Sam drinks beer', '35 years old'])

In [131]: dictionary_2.update({"age": "42"})

In [132]: dictionary_2
Out[132]: {'Sam': 'Sam drinks beer', 'age': '42'}
```

... class exercise



# Language specifics

- Assignment operators, data types, containers
  - Byte arrays:
    - they **store binary data** and may be part of a **data file**, **image file**, **compressed file**, downloaded server response, or many other files
    - they are **similar to python's string objects** used in Python 2.x version
    - the **important difference** is that **strings** are immutable and **byte arrays** are mutable
    - some applications make **lots of changes in large sets of memory** (such as image library or any other database) and **strings** perform very poorly in this kind of scenarios because a copy of the string in memory must be made, which takes unnecessary resources
    - **byte arrays** are better to be used when this kind of change is needed **without making a copy in memory**
    - rule: the immutable types **string** or **byte** should be used by default, unless the features described above are needed, then **byte arrays** come handy

# Language specifics

- Assignment operators, data types, containers

- Memory allocation and usage:

It is a good idea to be aware of the amount of memory used in your program (given in bytes):

```
Python
In [133]: from sys import getsizeof
In [134]: getsizeof(int)
Out[134]: 400
In [135]: getsizeof(True)
Out[135]: 28
In [136]: getsizeof(None)
Out[136]: 16
In [137]: getsizeof(NotImplemented)
Out[137]: 16
In [138]: getsizeof(float)
Out[138]: 400
In [139]: getsizeof(complex)
Out[139]: 400
In [140]: getsizeof(bytes)
Out[140]: 400
In [141]: getsizeof(dictionary_1)
Out[141]: 288
In [142]: getsizeof(dictionary_2)
Out[142]: 288
In [143]: whos
Variable      Type              Data/Info
-----
dictionary_1  dict              n=3
dictionary_2  dict              n=2
getsizeof     builtin_function_or_method
name          str               <built-in function getsizeof>
__loader__
```

... try it in class

# Language specifics

- Control flow (if/elif/else)
  - it is known as **control flow** as it controls the order in which the code is executed
  - the **'if'** statement is a conditional statement as it depends on certain condition(s) to execute some code
  - it comes with **'elif'** for a different condition and with **'else'** as default condition in case no other previously specified condition is met
  - the **'if'** statement in Python **does not require 'end'** to finish the statement

Example:

```
73 ## An example of an 'if' statement:  
74 x = 5  
75 if x == 2:  
76     print('x is 2')  
77 elif x == 4:  
78     print('x is 4')  
79 else:  
80     print('x is different than 2 or 4')  
81
```

... will result in:

```
...:  
x is different than 2 or 4
```

... class exercise

# Language specifics

- Conditional expressions
  - There are 4 basic ways you can test the validity of an expression easily:
    - `a == b`:
      - this expression tests **equality** between objects
      - Example:

```
In [144]: 34. == 34
Out[144]: True
In [145]: 34. == 35
Out[145]: False
```
    - `a in b`:
      - this checks if 'a' **exists** in the collection 'b'
      - Example:

```
In [146]: x = [41, 12, 'Alex', 34, 52]
In [147]: 'Alex' in x
Out[147]: True
In [148]: 35 in x
Out[148]: False
```
  - \* Note: if 'b' is a dictionary the test verifies that 'a' is an existing key in 'b'

# Language specifics

- Conditional expressions (cont.)
  - There are 4 basic ways you can test the validity of an expression easily:
    - a is b:
      - this expression check if both sides are **identical**

Example:

```
In [149]: 34. is 34
Out[149]: False
```

  - if <obj>:
    - this returns '**False**' when:
      - » the object is **<False>** or **<None>**
      - » the object is an **empty container** (*string, tuple, list, dictionary, etc.*)
      - » any number **equal to zero** (*0, 0.0 or 0+0j*)
    - it returns '**True**' **any other time**

Example:

```
In [150]: a=numpy.array([[3,5,4],[7,2,5]])
In [151]: if a.any() == True: e=5;
```