Object Oriented Programming II

Dr. Burak Kaleci

April 10, 2020

Content

Introduction

- NumPy (Numerical Python) is an open source Python library that's used in almost every field of science and engineering.
- It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems.
- NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development.
- The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

Source: This lecture note was prepared using the content at https://numpy.org/

Introduction

- The NumPy library contains multidimensional array and matrix data structures.
- It provides **ndarray**, a **homogeneous** (same data type) n-dimensional array object, with methods to efficiently operate on it. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.
- NumPy can be used to perform a wide variety of mathematical operations on arrays such as mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Introduction

There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the **same data type**, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

Introduction

There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.
- In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

Why is NumPy Fast?

- Consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

```
c=[]
for i in range(len(a)):
    c.append(a[i]*b[i])
```

- This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python.

Why is NumPy Fast?

- We could accomplish the same task much more quickly in C by writing:

```
for (i = 0; i ¡ rows; i++) {
    c[i] = a[i]*b[i];
}
```

- This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python.

Why is NumPy Fast?

- NumPy gives us the best of both worlds: element-by-element operations are the default mode when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy:

  c = a * b

- does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPys features which are the basis of much of its power: **vectorization** and **broadcasting**.

Why is NumPy Fast?

- **Vectorization** describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in optimized, pre-compiled C code.
- **Broadcasting** is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast.

How to import NumPy?

- In order to start using NumPy and all of the functions available in NumPy, you'll need to import it.
- This can be easily done with this import statement:

  import **numpy** as **np**
- Generally, we shorten **numpy** to **np** in order to save time and also to keep code standardized so that anyone working with your code can easily understand and run it.

What is an array?

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3, 4, 5, 6])

In [3]: a.dtype
Out[3]: dtype('int32')

In [4]: a.shape
Out[4]: (6,)

In [5]: b=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [6]: b.dtype
Out[6]: dtype('int32')

In [7]: b.shape
Out[7]: (3, 4)

In [8]: print(b[0])
[1 2 3 4]

In [9]: print(a[4])
5
```

- An array is a central data structure of the NumPy library.
- An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element.
- It has a grid of elements that can be indexed in various ways such as indexing, slicing and iterating.
- One way we can initialize NumPy arrays is from Python lists, using nested lists for two-or higher-dimensional data.

What is an array?

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3, 4, 5, 6])

In [3]: a.dtype
Out[3]: dtype('int32')

In [4]: a.shape
Out[4]: (6,)

In [5]: b=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [6]: b.dtype
Out[6]: dtype('int32')

In [7]: b.shape
Out[7]: (3, 4)

In [8]: print(b[0])
[1 2 3 4]

In [9]: print(a[4])
5
```

- The elements are all of the same type, referred to as the array **dtype**.
- The **shape** of the array is a tuple of integers giving the size of the array along each dimension.
- We can access the elements in the array using square brackets.
- When you're accessing elements, remember that indexing in NumPy starts at 0.
- That means that if you want to access the first element in your array, you'll be accessing element "0".

How to create a basic array?

- To create a NumPy array, you can use the function **np.array()** as used before:

  a = np.array([1, 2, 3])

- Besides creating an array from a sequence of elements, you can easily create an array filled with 0's:

  np.zeros(2)

- Or an array filled with 1s:

  np.ones(10)

- Or even an empty array! The function empty creates an array whose initial content is random and depends on the state of the memory:

  np.empty(6)

How to create a basic array?

- You can create an array with a range of elements:
  np.arange(4)
- And even an array that contains a range of evenly spaced intervals. To do this, you will specify the first number, last number, and the step size:
  np.arange(2, 9, 2)
- You can also use np.linspace() to create an array with values that are spaced linearly in a specified interval:
  np.linspace(0, 10, num=5)
- While the default data type is floating point (np.float64), you can explicitly specify which data type you want using the **dtype** keyword:
  np.ones(2, dtype=np.int64)

Adding, removing, and sorting elements

- Sorting an element is simple with np.sort(). If you start with this array: arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
- You can quickly sort the numbers in ascending order with:

  np.sort(arr)
- You can specify the axis, kind, and order when you call the function.

```
In [1]: import numpy as np

In [2]: b=np.array([[7,1,4,12],[2,9,8,3]])

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

In [4]: np.sort(b,axis=0,kind='mergesort')
Out[4]:
array([[ 2,  1,  4,  3],
       [ 7,  9,  8, 12]])

In [5]: np.sort(b,axis=1,kind='heapsort')
Out[5]:
array([[ 1,  4,  7, 12],
       [ 2,  3,  8,  9]])

In [6]: np.sort(b,axis=0,kind='quicksort')[::-1]
Out[6]:
array([[ 7,  9,  8, 12],
       [ 2,  1,  4,  3]])

In [7]: np.sort(b,axis=1,kind='quicksort')[::-1]
Out[7]:
array([[ 2,  3,  8,  9],
       [ 1,  4,  7, 12]])
```

You can find detailed information at https://numpy.org/devdocs/reference/generated/numpy.sort

Adding, removing, and sorting elements

- You can concatenate them with np.concatenate()
- In order to remove elements from an array, it's simple to use indexing to select the elements that you want to keep.

```
In [1]: import numpy as np

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

In [3]: b = np.array([5, 6, 7, 8])

In [4]: np.concatenate((a, b))
Out[4]: array([1, 2, 3, 4, 5, 6, 7, 8])

In [5]: x = np.array([[1, 2], [3, 4]])

In [6]: y = np.array([[5, 6]])

In [7]: np.concatenate((x, y), axis=0)
Out[7]:
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Shape and size of an array

```
In [1]: import numpy as np

In [2]: array_example = np.array([[[0, 1, 2, 3],
   ...:                            [4, 5, 6, 7]],
   ...:                           [[0, 1, 2, 3],
   ...:                            [4, 5, 6, 7]],
   ...:                          [[0 ,1 ,2, 3],
   ...:                           [4, 5, 6, 7]]])

In [3]: array_example.ndim
Out[3]: 3

In [4]: array_example.size
Out[4]: 24

In [5]: array_example.shape
Out[5]: (3, 2, 4)
```

- **ndim** will tell you the number of axes, or dimensions, of the array.
- **size** will tell you the total number of elements of the array. This is the product of the elements of the array's shape.
- **shape** will display a tuple of integers that indicate the number of elements stored along each dimension of the array.

Reshape of an array

```
In [1]: import numpy as np

In [2]: a = np.arange(6)

In [3]: print(a)
[0 1 2 3 4 5]

In [4]: b = a.reshape(3, 2)

In [5]: print(b)
[[0 1]
 [2 3]
 [4 5]]
```

- Using **arr.reshape()** will give a new shape to an array without changing the data.
- Just remember that when you use the reshape method, the array you want to produce needs to have the same number of elements as the original array.
- **If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.**

How to add a new axis to an array

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3, 4, 5, 6])

In [3]: a.shape
Out[3]: (6,)

In [4]: row_vector = a[np.newaxis, :]
   ...: row_vector.shape
Out[4]: (1, 6)

In [5]: col_vector = a[:, np.newaxis]

In [6]: col_vector.shape
Out[6]: (6, 1)

In [7]: b = np.expand_dims(a, axis=1)

In [8]: b.shape
Out[8]: (6, 1)

In [9]: c = np.expand_dims(a, axis=0)

In [10]: c.shape
Out[10]: (1, 6)

In [11]: d=np.expand_dims(b,axis=2)

In [12]: d.shape
Out[12]: (6, 1, 1)
```

- You can use **newaxis** and **expand_dims** to increase the dimensions of your existing array.
- This means that a 1D array will become a 2D array, a 2D array will become a 3D array, and so on.
- You can explicitly convert a 1D array with either a **row vector** or a **column vector** using **newaxis**.
- For example, you can convert a 1D array to a row vector by inserting an axis along the first dimension or for a column vector, you can insert an axis along the second dimension.
- You can also expand an array by inserting a new axis at a specified position with **expand_dims**.

Indexing and slicing

- You can index and slice NumPy arrays in the same ways you can slice Python lists.

```
In [14]: data = np.array([1, 2, 3])

In [15]: data[1]
Out[15]: 2

In [16]: data[0:2]
Out[16]: array([1, 2])

In [17]: data[1:]
Out[17]: array([2, 3])

In [18]: data[-2:]
Out[18]: array([2, 3])
```

- You may want to take a section of your array or specific array elements to use in further analysis or additional operations. To do that, you'll need to subset, slice, and/or index your arrays.
- If you want to select values from your array that fulfill certain conditions, it's straightforward with NumPy.
- You can easily print all of the values in the array that are less than 5.
- You can also select, for example, numbers that are equal to or greater than 5, and use that condition to index an array.

```
In [20]: a = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [21]: print(a[a < 5])
[1 2 3 4]

In [22]: five_up = (a >= 5)

In [23]: print(a[five_up])
[ 5  6  7  8  9 10 11 12]
```

Indexing and slicing

- You can select elements that are divisible by 2.
- You can select elements that satisfy two conditions using the & and | operators.
- You can also make use of the logical operators & and | in order to return boolean values that specify whether or not the values in an array fulfill a certain condition. This can be useful with arrays that contain names or other categorical values.

```
In [25]: divisible_by_2 = a[a%2==0]

In [26]: print(divisible_by_2)
[ 2  4  6  8 10 12]

In [27]: c = a[(a > 2) & (a < 11)]

In [28]: print(c)
[ 3  4  5  6  7  8  9 10]

In [29]: five_up = (a > 5) | (a == 5)

In [30]: print(five_up)
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]]
```

Indexing and slicing

- You can also use **nonzero()** to select elements or indices from an array.
- You can use nonzero() to print the indices of elements that are, for example, less than 5.
- In this example, a tuple of arrays was returned: one for each dimension. The **first array** represents the **row indices** where these values are found, and the **second array** represents the **column indices** where the values are found.

```
In [36]: a=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [37]: b = np.nonzero(a < 5)

In [38]: print(b)
(array([0, 0, 0, 0], dtype=int64), array([0, 1, 2, 3], dtype=int64))

In [39]: list_of_coordinates= list(zip(b[0], b[1]))

In [40]: for coord in list_of_coordinates:
    ...:     print(coord)
(0, 0)
(0, 1)
(0, 2)
(0, 3)

In [41]: print(a[b])
[1 2 3 4]
```

Indexing and slicing

- If you want to generate a list of coordinates where the elements exist, you can zip the arrays, iterate over the list of coordinates, and print them.
- You can also use nonzero() to print the elements in an array that are less than 5.
- If the element you're looking for doesn't exist in the array, then the returned array of indices will be empty.

```
In [36]: a=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [37]: b = np.nonzero(a < 5)

In [38]: print(b)
(array([0, 0, 0, 0], dtype=int64), array([0, 1, 2, 3], dtype=int64))

In [39]: list_of_coordinates= list(zip(b[0], b[1]))

In [40]: for coord in list_of_coordinates:
    ...:     print(coord)
(0, 0)
(0, 1)
(0, 2)
(0, 3)

In [41]: print(a[b])
[1 2 3 4]
```

How to create an array from existing data

- You can stack two existing arrays, both vertically (vstack) and horizontally (hstack).

```
In [45]: a1=np.array([[1,1],[2,2]])

In [46]: a2=np.array([[3,3],[4,4]])

In [47]: np.vstack((a1, a2))
Out[47]:
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])

In [48]: np.hstack((a1, a2))
Out[48]:
array([[1, 1, 3, 3],
       [2, 2, 4, 4]])
```

- You can split an array into several smaller arrays using hsplit. You can specify either the number of equally shaped arrays to return or the columns after which the division should occur.

```
In [49]: x = np.arange(1, 25).reshape(2, 12)

In [50]: print(x)
[[ 1  2  3  4  5  6  7  8  9 10 11 12]
 [13 14 15 16 17 18 19 20 21 22 23 24]]

In [51]: np.hsplit(x, 3)
Out[51]:
[array([[ 1,  2,  3,  4],
        [13, 14, 15, 16]]), array([[ 5,  6,  7,  8],
        [17, 18, 19, 20]]), array([[ 9, 10, 11, 12],
        [21, 22, 23, 24]])]

In [52]: np.hsplit(x, (3, 4))
Out[52]:
[array([[ 1,  2,  3],
        [13, 14, 15]]), array([[ 4],
        [16]]), array([[ 5,  6,  7,  8,  9, 10, 11, 12],
        [17, 18, 19, 20, 21, 22, 23, 24]])]
```

How to create an array from existing data

- You can use the **view** method to create a new array object that looks at the same data as the original array **(a shallow copy)**.
- Views are an important NumPy concept! NumPy functions, as well as operations like indexing and slicing, will return views whenever possible.
- This saves memory and is faster (no copy of the data has to be made).
- However it's important to be aware of this modifying data in a view also modifies the original array!
- Using the copy method will make a complete copy of the array and its data **(a deep copy)**.

```
In [54]: a=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [55]: b1 = a[0, :]

In [56]: print(b1)
[1 2 3 4]

In [57]: b1[0] = 99

In [58]: print(b1)
[99  2  3  4]

In [59]: print(a)
[[99  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

In [60]: b2 = a.copy()

In [61]: b2[1] = 1000

In [62]: print(a)
[[99  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Basic array operations

- You can perform basic operations on the arrays together with the traditional notation.
- Basic operations are simple with NumPy.
- If you want to find the sum of the elements in an array, you'd use **sum()**. This works for 1D arrays, 2D arrays, and arrays in higher dimensions.
- To add the rows or the columns in a 2D array, you would specify the axis.

```
In [2]: data = np.array([1, 2])

In [3]: ones = np.ones(2, dtype=int)

In [4]: data + ones
Out[4]: array([2, 3])

In [5]: data - ones
Out[5]: array([0, 1])

In [6]: data * data
Out[6]: array([1, 4])

In [7]: data / data
Out[7]: array([1., 1.])

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

In [9]: a.sum()
Out[9]: 10

In [10]: b = np.array([[1, 1], [2, 2]])

In [11]: b.sum(axis=0)
Out[11]: array([3, 3])

In [12]: b.sum(axis=1)
Out[12]: array([2, 4])
```

Basic array operations

- There are times when you might want to carry out an operation between an array and a single number (also called an operation between a vector and a scalar) or between arrays of two different sizes.
- For example, your array (we'll call it "data") might contain information about distance in miles but you want to convert the information to kilometers. You can perform this operation with:

```
data = np.array([1.0, 2.0])
data * 1.6
array([1.6, 3.2])
```

Basic array operations

- **NumPy understands that the multiplication should happen with each cell.**
- That concept is called **broadcasting**. Broadcasting is a mechanism that allows NumPy to perform operations on arrays of different shapes.
- The dimensions of your array must be compatible, for example, when the dimensions of both arrays are equal or when one of them is 1.
- If the dimensions are not compatible, you will get a ValueError.

Basic array operations

- NumPy also performs aggregation functions. In addition to **min**, **max**, and **sum**, you can easily run **mean** to get the average, **prod** to get the result of multiplying the elements together, **std** to get the standard deviation, and more.
- By default, every NumPy aggregation function will return the aggregate of the entire array.
- It's very common to want to aggregate along a row or column. You can specify on which axis you want the aggregation function to be computed.

```
In [15]: a = np.array([[0.45053314, 0.17296777, 0.34376245, 0.5510652],
    ...:               [0.54627315, 0.05093587, 0.40067661, 0.55645993],
    ...:               [0.12697628, 0.82485143, 0.26590556, 0.56917101]])

In [16]: a.sum()
Out[16]: 4.8595784

In [17]: a.min()
Out[17]: 0.05093587

In [18]: a.max()
Out[18]: 0.82485143

In [19]: a.min(axis=0)
Out[19]: array([0.12697628, 0.05093587, 0.26590556, 0.5510652 ])
```

Generating random numbers

- The use of random number generation is an important part of the configuration and evaluation of many numerical and machine learning algorithms.
- Whether you need to randomly initialize weights in an artificial neural network, split data into random sets, or randomly shuffle your dataset, being able to generate random numbers (actually, repeatable pseudo-random numbers) is essential.
- You can generate a 2 x 4 array of random integers between 0 and 4 with:

```
rng.integers(5, size=(2, 4))
array([[2, 1, 1, 0],
       [0, 0, 0, 4]])
```

How to get unique items and counts

- You can find the unique elements in an array easily with **unique**.
- To get the indices of unique values in a NumPy array (an array of first index positions of unique values in the array), just pass the **return_index argument** in np.unique() as well as your array.
- You can pass the r**eturn_counts argument** in np.unique() along with your array to get the frequency count of unique values in a NumPy array.

```
In [21]: a=np.array([11,11,12,13,14,15,16,17,12,13,11,14,18,19,20])

In [22]: unique_values = np.unique(a)

In [23]: print(unique_values)
[11 12 13 14 15 16 17 18 19 20]

In [24]: unique_values, indices_list = np.unique(a, return_index=True)

In [25]: print(indices_list)
[ 0  2  3  4  5  6  7 12 13 14]

In [26]: unique_values, occurrence_count = np.unique(a, return_counts=True)

In [27]: print(occurrence_count)
[3 2 2 2 1 1 1 1 1 1]
```

How to get unique items and counts

- You can find unique values with 2D arrays.
- If the axis argument isn't passed, your 2D array will be flattened.
- If you want to get the unique rows or columns, make sure to pass the axis argument. To find the unique rows, specify axis=0 and for columns, specify axis=1.

```
In [29]: a_2d=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[1,2,3,4]])

In [30]: unique_values = np.unique(a_2d)

In [31]: print(unique_values)
[ 1  2  3  4  5  6  7  8  9 10 11 12]

In [32]: unique_rows = np.unique(a_2d, axis=0)

In [33]: print(unique_rows)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

In [34]: unique_rows, indices, occurrence_count = np.unique(a_2d, axis=0,
return_counts=True, return_index=True)

In [35]: print(indices)
[0 1 2]

In [36]: print(occurrence_count)
[2 1 1]
```

Transposing and reshaping a matrix

- It's common to need to transpose your matrices. NumPy arrays have the property **T** that allows you to transpose a matrix.
- You can also use **transpose** to reverse or change the axes of an array according to the values you specify.
- You may also need to switch the dimensions of a matrix.
- This can happen when, for example, you have a model that expects a certain input shape that is different from your dataset.
- This is where the **reshape** method can be useful.

```
In [44]: data=np.array([[1,2],[3,4],[5,6]])

In [45]: print(data)
[[1 2]
 [3 4]
 [5 6]]

In [46]: data.T
Out[46]:
array([[1, 3, 5],
       [2, 4, 6]])

In [47]: data.transpose()
Out[47]:
array([[1, 3, 5],
       [2, 4, 6]])

In [48]: data.reshape(2, 3)
Out[48]:
array([[1, 2, 3],
       [4, 5, 6]])

In [49]: data.reshape(3, 2)
Out[49]:
array([[1, 2],
       [3, 4],
       [5, 6]])
```

How to reverse an array

- NumPys **flip()** function allows you to flip, or reverse, the contents of an array along an axis.
- When using **flip**, specify the array you would like to reverse and the axis. If you don't specify the axis, NumPy will reverse the contents along all of the axes of your input array.
- You can also reverse the contents of only one column or row.

```
In [87]: arr_2d=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [88]: print(arr_2d)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

In [89]: reversed_arr_rows = np.flip(arr_2d, axis=0)

In [90]: print(reversed_arr_rows)
[[ 9 10 11 12]
 [ 5  6  7  8]
 [ 1  2  3  4]]

In [91]: np.flip(arr_2d, axis=1)
Out[91]:
array([[ 4,  3,  2,  1],
       [ 8,  7,  6,  5],
       [12, 11, 10,  9]])

In [92]: arr_2d[:,1] = np.flip(arr_2d[:,1],axis=0)

In [93]: print(arr_2d)
[[ 1 10  3  4]
 [ 5  6  7  8]
 [ 9  2 11 12]]

In [94]: arr_2d[1] = np.flip(arr_2d[1],axis=0)

In [95]: print(arr_2d)
[[ 1 10  3  4]
 [ 8  7  6  5]
 [ 9  2 11 12]]
```

Reshaping and flattening multidimensional arrays

- There are two popular ways to flatten an array: **flatten()** and **ravel()**.
- The primary difference between the two is that the new array created using ravel() is actually a reference to the parent array (i.e., a "view").
- This means that any changes to the new array will affect the parent array as well. Since ravel does not create a copy, it's memory efficient.
- When you use flatten, changes to your new array won't change the parent array.

```
In [97]: x=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,1

In [98]: a1 = x.flatten()

In [99]: a1[0] = 99

In [100]: print(x)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

In [101]: print(a1)
[99  2  3  4  5  6  7  8  9 10 11 12]

In [102]: a2 = x.ravel()

In [103]: a2[0] = 98

In [104]: print(x)
[[98  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

In [105]: print(a2)
[98  2  3  4  5  6  7  8  9 10 11 12]
```

How to save and load NumPy objects

- You will want to save your arrays to disk and load them back without having to re-run the code.
- Fortunately, there are several ways to save and load objects with NumPy.
- The ndarray objects can be saved to and loaded from the disk files with **loadtxt** and **savetxt** functions that handle normal text files, **load** and **save** functions that handle NumPy binary files with a .npy file extension, and a **savez** function that handles NumPy files with a .npz file extension.
- The .npy and .npz files store data, shape, dtype, and other information required to reconstruct the ndarray in a way that allows the array to be correctly retrieved, even when the file is on another machine with different architecture.
- If you want to store a single ndarray object, store it as a .npy file using np.save.
- If you want to store more than one ndarray object in a single file, save it as a .npz file using np.savez.
- You can also save several arrays into a single file in compressed npz format with savez_compressed.

How to save and load NumPy objects

- It's easy to save and load and array with **save()**. Just make sure to specify the array you want to save and a file name.
- For example, if you create this array:

  a = np.array([1, 2, 3, 4, 5, 6])
- You can save it as "filename.npy" with:

  np.save('filename', a)
- You can use **load()** to reconstruct your array

  b = np.load('filename.npy')
- You can save a NumPy array as a plain text file like a .csv or .txt file with **savetxt()**:

  np.savetxt('new_file.csv', a)
- You can quickly and easily load your saved text file using **loadtxt()**:

  np.loadtxt('new_file.csv')

Introduction

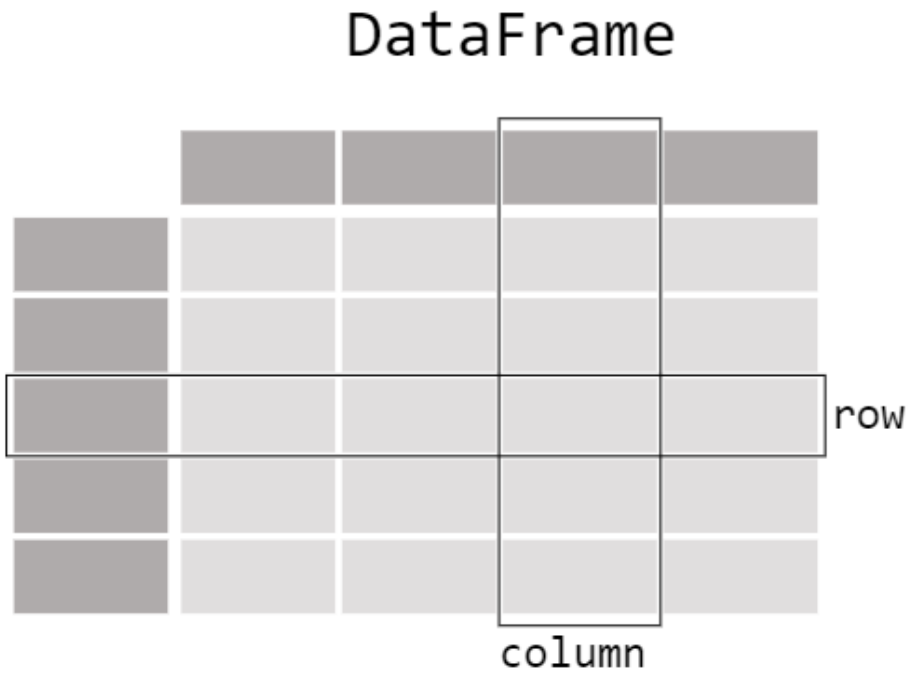
- pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
- To load the pandas package and start working with it, import the package.

  import **pandas** as **pd**
- The community agreed alias for pandas is pd, so loading pandas as pd is assumed standard practice for all of the pandas documentation.

Source: This lecture note was prepared using the content at https://pandas.pydata.org/

DataFrame

- A **DataFrame** is a 2-dimensional data structure that can store data of different types (including characters, integers, floating point values, categorical data and more) in columns. It is similar to a spreadsheet, a SQL table or the data.frame in R.
- To manually store data in a table, create a DataFrame. When using a Python dictionary of lists, the dictionary **keys** will be used as **column headers** and the **values** in each list as **rows** of the DataFrame.



DataFrame

- You can store passenger data of the Titanic. For a number of passengers, you may know the name (characters), age (integers) and sex (male/female) data.

```
In [1]: import pandas as pd

In [2]: df=pd.DataFrame({"Name": ["Braund, Mr. Owen Harris",
   ...: "Allen, Mr. William Henry",
   ...: "Bonnell, Miss. Elizabeth"],
   ...: "Age": [22, 35, 58],
   ...: "Sex": ["male", "male", "female"]}
   ...: )

In [3]: df
Out[3]:
                         Name  Age     Sex
0   Braund, Mr. Owen Harris   22    male
1  Allen, Mr. William Henry   35    male
2  Bonnell, Miss. Elizabeth   58  female
```

- The table has 3 columns, each of them with a column label. The column labels are respectively Name, Age and Sex.
- The column Name consists of textual data with each value a string, the column Age are numbers and the column Sex is textual data.

Series

- **Each column in a DataFrame is a Series.** So, when selecting a single column of a pandas DataFrame, the result is a pandas Series.
- To select the column, use the column label in between square brackets [].
- For example, you can intend to work with the data in the column Age:

  df["Age"]
- You can create a Series from scratch as well:

  ages = pd.Series([22, 35, 58], name="Age")
- A pandas Series has no column labels, as it is just a single column of a DataFrame. A Series does have row labels.

Series

- You can find the maximum age of the passengers:
  df["Age"].max()
- As illustrated by the max() method, you can do things with a DataFrame or Series. pandas provides a lot of functionalities, each of them a method you can apply to a DataFrame or Series.
- The **describe()** method provides a quick overview of the numerical data in a DataFrame. As the Name and Sex columns are textual data, these are by default not taken into account by the describe() method.

How do I read and write tabular data?

- pandas provides the **read_csv()** function to read data stored as a csv file into a pandas DataFrame.

  titanic = pd.read_csv("titanic.csv")
- pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet,), each of them with **the prefix read_***.
- You can use the **head(N)** method to see the first N rows of a DataFrame.
- Similarly, you can use **tail(N)** method to see the last N rows of a DataFrame.
- The method **info()** provides technical information about a DataFrame.

How do I read and write tabular data?

- Whereas **read_*** functions are used to read data to pandas, the **to_*** methods are used to store data.

  titanic.to_excel('titanic.xlsx', sheet_name='passengers', index=False)
- The **to_excel()** method stores the data as an excel file.
- In the example here, the **sheet_name** is named passengers instead of the default Sheet1.
- By setting index=False the row index labels are not saved in the spreadsheet.
- The equivalent read function to_excel() will reload the data to a DataFrame:

  titanic = pd.read_excel('titanic.xlsx', sheet_name= 'passengers')

How do I select specific columns from a DataFrame?

- Assume that, you will use the titanic data for examples.
- **To select a single column, use square brackets [] with the column name of the column of interest.**
- **Each column in a DataFrame is a Series**. As a single column is selected, the returned object is a pandas DataFrame.
- A pandas Series is 1-dimensional and only the number of rows is returned.
- **To select multiple columns, use a list of column names within the selection brackets []**.
- To select multiple columns, use a list of column names within the selection brackets []

How do I select specific columns from a DataFrame?

```
In [10]: import pandas as pd

In [11]: titanic = pd.read_csv("titanic.csv")

In [12]: titanic.head()
Out[12]:
   PassengerId  Survived     Pclass     ...         Fare Cabin  Embarked
1            0         3     Braund     ...       7.2500   NaN         S
2            1         1    Cumings     ...      71.2833   C85         C
3            1         3  Heikkinen     ...       7.9250   NaN         S
4            1         1   Futrelle     ...      53.1000  C123         S
5            0         3      Allen     ...       8.0500   NaN         S

[5 rows x 12 columns]

In [13]: ages = titanic["Age"]

In [14]: ages.head()
Out[14]:
1    22.0
2    38.0
3    26.0
4    35.0
5    35.0
Name: Age, dtype: float64

In [15]: titanic["Age"].shape
Out[15]: (891,)

In [16]: age_sex = titanic[["Age", "Sex"]]

In [17]: age_sex.shape
Out[17]: (891, 2)
```

How do I filter specific rows from a DataFrame?

- To select rows based on a conditional expression, use a condition inside the selection brackets [].
- For example,you can intend to obtain the passengers older than 35 years:
  $above\_35 = titanic[titanic["Age"] > 35]$
- The condition inside the selection brackets $titanic["Age"] > 35$ checks for which rows the Age column has a value larger than 35
- The output of the conditional expression ($>$, but also $==, !=, <, <=,$ would work) is actually a pandas Series of boolean values (either True or False) with the same number of rows as the original DataFrame.
- Such a Series of boolean values can be used to filter the DataFrame by putting it in between the selection brackets []. Only rows for which the value is True will be selected.
- We now from before that the original titanic DataFrame consists of 891 rows. Let's have a look at the amount of rows which satisfy the condition by checking the shape attribute of the resulting DataFrame above_35.

How do I filter specific rows from a DataFrame?

- Similar to the conditional expression, the **isin()** conditional function returns a True for each row the values are in the provided list.
- To filter the rows based on such a function, use the conditional function inside the selection brackets [].

  class_23 = titanic[titanic["Pclass"].isin([2, 3])]
- In this case, the condition inside the selection brackets titanic["Pclass"].isin([2, 3]) checks for which rows the Pclass column is either 2 or 3.
- The above is equivalent to filtering by rows for which the class is either 2 or 3 and combining the two statements with an | (or) operator:

  class_23 = titanic[(titanic["Pclass"] == 2) | (titanic["Pclass"] == 3)]
- The **notna()** conditional function returns a True for each row the values are not an Null value. As such, this can be combined with the selection brackets [] to filter the data table.

  age_no_na = titanic[titanic["Age"].notna()]

How do I select specific rows and columns from a DataFrame?

- In this case, a subset of both rows and columns is made in one go and just using selection brackets [] is not sufficient anymore.
- The **loc/iloc** operators are required in front of the selection brackets [].
- When using **loc/iloc**, the part before the comma is the rows you want, and the part after the comma is the columns you want to select.
- For example, the following statement selects the names of the passengers older than 35 years:

  adult_names = titanic.loc[titanic["Age"] ¿ 35, "Name"]
- When using the column names, row labels or a condition expression, use the **loc** operator in front of the selection brackets []. For both the part before and after the comma, you can use a single label, a list of labels, a slice of labels, a conditional expression or a colon. Using a colon specificies you want to select all rows or columns.

How do I select specific rows and columns from a DataFrame?

- Again, a subset of both rows and columns is made in one go and just using selection brackets [] is not sufficient anymore.
- When specifically interested in certain rows and/or columns based on their position in the table, use the **iloc** operator in front of the selection brackets [].
- For example, the following statement selects the rows 9 till 25 and columns 3 to 5:

  titanic.iloc[9:25, 2:5]
- When selecting specific rows and/or columns with loc or iloc, new values can be assigned to the selected data. For example, to assign the name anonymous to the first 3 elements of the third column:

  titanic.iloc[0:3, 3] = "anonymous"

How to create new columns derived from existing columns?

- **To create a new column, use the [] brackets with the new column name at the left side of the assignment.**

```
air_quality["london_mg_per_cubic"] =
air_quality["station_london"] * 1.882
```

- **The calculation of the values is done element_wise.** This means all values in the given column are multiplied by the value 1.882 at once. **You do not need to use a loop to iterate each of the rows!**
- **Also mathematical operators** $(+, -, *, /)$ **or logical operators** $(<, >, =,)$ **work element wise.**
- The **rename()** function can be used for both row labels and column labels. Provide a dictionary with the keys the current names and the values the new names to update the corresponding names.

How to calculate summary statistics?

- Different statistics are available and can be applied to columns with numerical data. Operations in general exclude missing data and operate across rows by default.
- For example,you can calculate the average age of the titanic passengers:
  titanic["Age"].mean()
- The statistic applied to multiple columns of a DataFrame is calculated for each numeric column.
- For example,you can calculate median age and ticket fare price of the titanic passengers:
  titanic[["Age", "Fare"]].median()
- The aggregating statistic can be calculated for multiple columns at the same time with **describe()** method.
- Instead of the predefined statistics, specific combinations of aggregating statistics for given columns can be defined using the **agg()** method.

How to calculate summary statistics?

```
In [25]: titanic["Age"].mean()
Out[25]: 29.69911764705882

In [26]: titanic[["Age", "Fare"]].median()
Out[26]:
Age     28.0000
Fare    14.4542
dtype: float64

In [27]: titanic[["Age", "Fare"]].describe()
Out[27]:
              Age        Fare
count  714.000000  891.000000
mean    29.699118   32.204208
std     14.526497   49.693429
min      0.420000    0.000000
25%     20.125000    7.910400
50%     28.000000   14.454200
75%     38.000000   31.000000
max     80.000000  512.329200

In [28]: titanic.agg({'Age': ['min', 'max', 'median', 'skew'],
    ...:              'Fare': ['min', 'max', 'median', 'mean']})
Out[28]:
              Age        Fare
max     80.000000  512.329200
mean          NaN   32.204208
median  28.000000   14.454200
min      0.420000    0.000000
skew     0.389108         NaN
```

Aggregating statistics grouped by category

- What is the average age for male versus female titanic passengers?
  titanic[["Sex", "Age"]].groupby("Sex").mean()
- As our interest is the average age for each gender, a subselection on these two columns is made first: titanic[["Sex", "Age"]].
- Next, the **groupby()** method is applied on the Sex column to make a group per category.
- The average age for each gender is calculated and returned.
- Calculating a given statistic (e.g. mean age) for each category in a column (e.g. male/female in the Sex column) is a common pattern.
- The **groupby** method is used to support this type of operations. More general, this fits in the more general split-apply-combine pattern:
  - Split the data into groups
  - Apply a function to each group independently
  - Combine the results into a data structure

Aggregating statistics grouped by category

- In the previous example, we explicitly selected the 2 columns first. If not, the mean method is applied to each column containing numerical columns:

  titanic.groupby("Sex").mean()

- It does not make much sense to get the average value of the Pclass. if we are only interested in the average age for each gender, the selection of columns (rectangular brackets [] as usual) is supported on the grouped data as well:

  titanic.groupby("Sex")["Age"].mean()

- Grouping can be done by multiple columns at the same time. Provide the column names as a list to the groupby() method:

  titanic.groupby(["Sex", "Pclass"])["Fare"].mean()

- The **value_counts()** method counts the number of records for each category in a column.

  titanic["Pclass"].value_counts()

Aggregating statistics grouped by category

```
In [30]: titanic[["Sex", "Age"]].groupby("Sex").mean()
Out[30]:
              Age
Sex
female  27.915709
male    30.726645

In [31]: titanic.groupby("Sex").mean()
Out[31]:
        PassengerId  Survived    ...         Parch       Fare
Sex                              ...
female     0.742038  2.159236    ...      0.649682  44.479818
male       0.188908  2.389948    ...      0.235702  25.523893

[2 rows x 6 columns]

In [32]: titanic.groupby("Sex")["Age"].mean()
Out[32]:
Sex
female    27.915709
male      30.726645
Name: Age, dtype: float64
```

How to reshape the layout of tables?

- With **sort_values()**, the rows in the table are sorted according to the defined column(s). **The index will follow the row order.**
- For example,you can sort the titanic data according to the age of the passengers

  titanic.sort_values(by="Age").head()
- You can also sort the titanic data according to the cabin class and age in descending order

  titanic.sort_values(by=['Pclass', 'Age'], ascending=False).head()

Introduction

- Matplotlib is a library for making 2D and 3D plots of arrays in Python. Although it has its origins in emulating the MATLAB graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way.
- Although Matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.
- To load the Matplotlib package and start working with it, import the package.

  import **matplotlib.pyplot** as **plt**

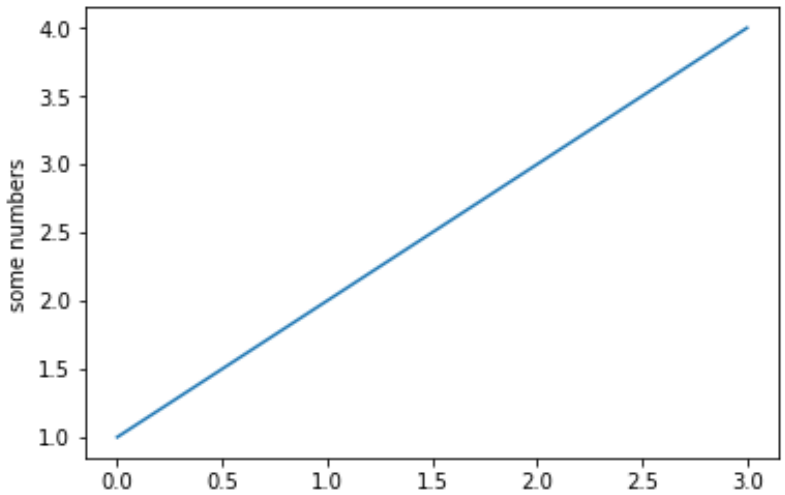
  Source: This lecture note was prepared using the content at https://matplotlib.org/index.html

Intro to pyplot

- **matplotlib.pyplot** is a collection of command style functions that make matplotlib work like MATLAB.
- Each **pyplot** function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- In matplotlib.pyplot various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes.

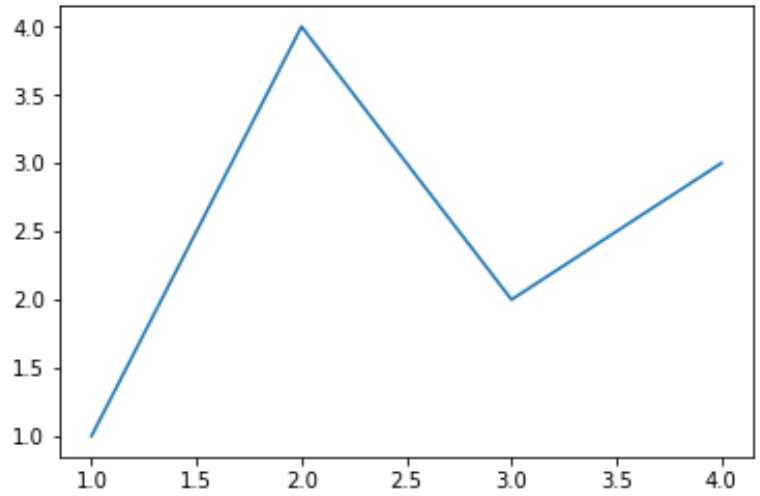
Simple Examples

```
In [13]: import matplotlib.pyplot as plt
    ...: plt.plot([1, 2, 3, 4])
    ...: plt.ylabel('some numbers')
    ...: plt.show()
```




```
In [1]: import matplotlib.pyplot as plt

In [2]: plt.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Mat
Out[2]: [<matplotlib.lines.Line2D at 0x2245b3a5f98
```

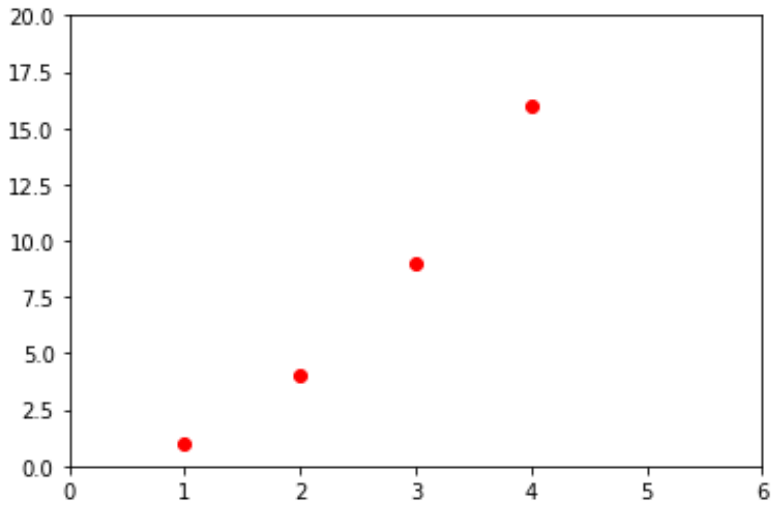


You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0, 1, 2, 3].

Simple Examples

- For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot.
- The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string.
- The default format string is 'b-', which is a solid blue line.
- The axis() command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

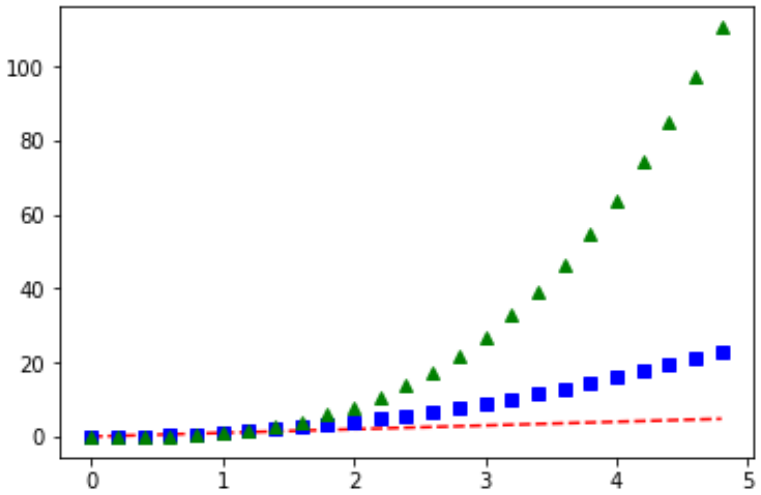
```
In [15]: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
    ...: plt.axis([0, 6, 0, 20])
    ...: plt.show()
```



Simple Examples

- If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally.

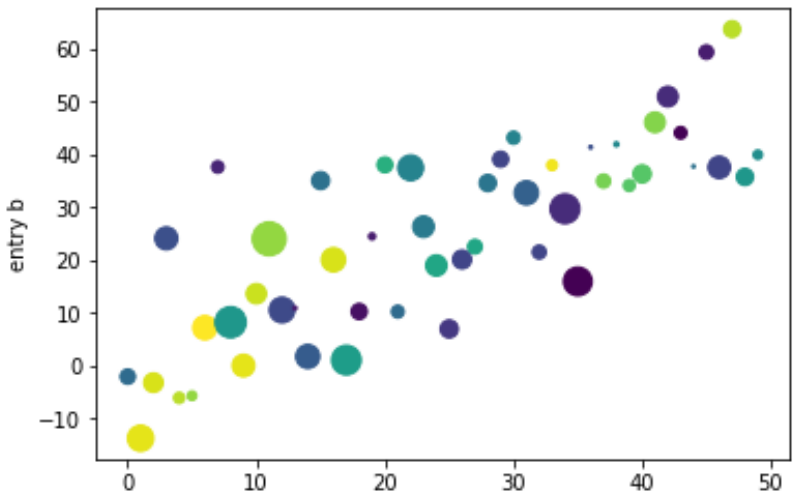
```
In [17]: import numpy as np
    ...:
    ...: # evenly sampled time at 200ms intervals
    ...: t = np.arange(0., 5., 0.2)
    ...:
    ...: # red dashes, blue squares and green triangles
    ...: plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
    ...: plt.show()
```



Plotting with keyword strings

- Matplotlib allows you provide such an object with the data keyword argument. If provided, then you may generate plots with the strings corresponding to these variables.

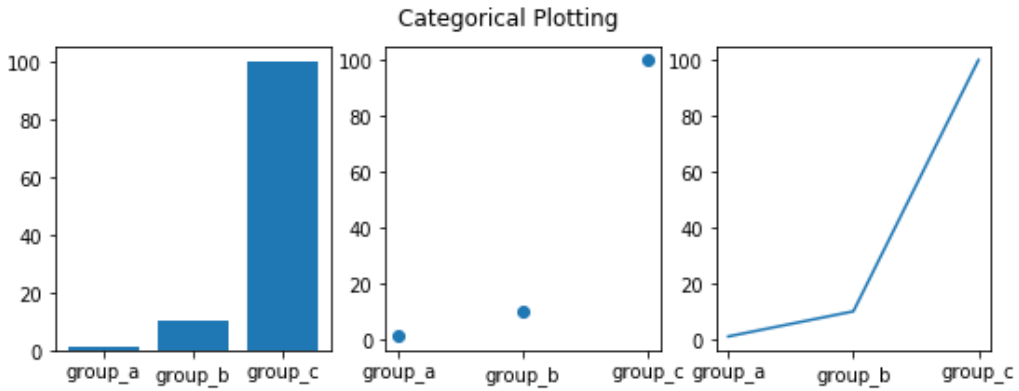
```
In [19]: data = {'a': np.arange(50),
    ...:          'c': np.random.randint(0, 50, 50),
    ...:          'd': np.random.randn(50)}
    ...: data['b'] = data['a'] + 10 * np.random.randn(50)
    ...: data['d'] = np.abs(data['d']) * 100
    ...:
    ...: plt.scatter('a', 'b', c='c', s='d', data=data)
    ...: plt.xlabel('entry a')
    ...: plt.ylabel('entry b')
    ...: plt.show()
```



Plotting with categorical variables

- It is also possible to create a plot using categorical variables. Matplotlib allows you to pass categorical variables directly to many plotting functions.

```
In [21]: names = ['group_a', 'group_b', 'group_c']
    ...: values = [1, 10, 100]
    ...:
    ...: plt.figure(figsize=(9, 3))
    ...:
    ...: plt.subplot(131)
    ...: plt.bar(names, values)
    ...: plt.subplot(132)
    ...: plt.scatter(names, values)
    ...: plt.subplot(133)
    ...: plt.plot(names, values)
    ...: plt.suptitle('Categorical Plotting')
    ...: plt.show()
```



Controlling line properties

- Lines have many attributes that you can set: linewidth, dash style, antialiased, etc. There are several ways to set line properties:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a Line2D instance. plot returns a list of Line2D objects; e.g., line1, line2 = plot(x1, y1, x2, y2). In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with line, to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False)
```

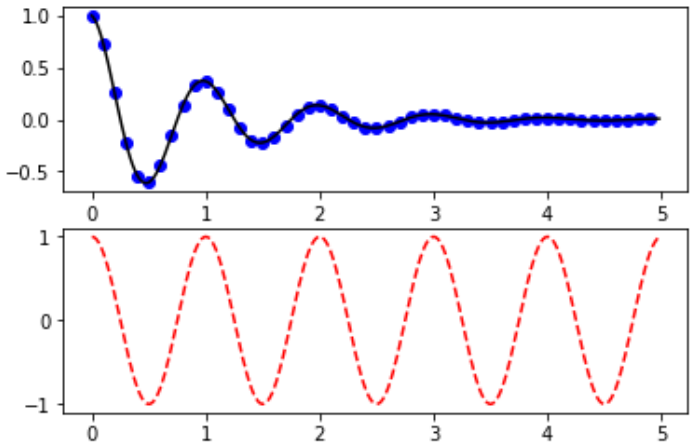
- Use the setp() command. The example below uses a MATLAB-style command to set multiple properties on a list of lines.

```
lines = plt.plot(x1, y1, x2, y2)
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Working with multiple figures and axes

- The figure() command here is optional because figure(1) will be created by default, just as a subplot(111) will be created by default if you don't manually specify any axes.
- The subplot() command specifies numrows, numcols, plot_number where plot_number ranges from 1 to *numrows* $*$ *numcols*. The commas in the subplot command are optional if *numrows* $*$ *numcols* < 10. So subplot(211) is identical to subplot(2, 1, 1).
- You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use the axes() command, which allows you to specify the location as axes([left, bottom, width, height]) where all values are in fractional (0 to 1) coordinates.

```
In [23]: def f(t):
    ...:     return np.exp(-t) * np.cos(2*np.pi*t)
    ...:
    ...: t1 = np.arange(0.0, 5.0, 0.1)
    ...: t2 = np.arange(0.0, 5.0, 0.02)
    ...:
    ...: plt.figure()
    ...: plt.subplot(211)
    ...: plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
    ...:
    ...: plt.subplot(212)
    ...: plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
    ...: plt.show()
```

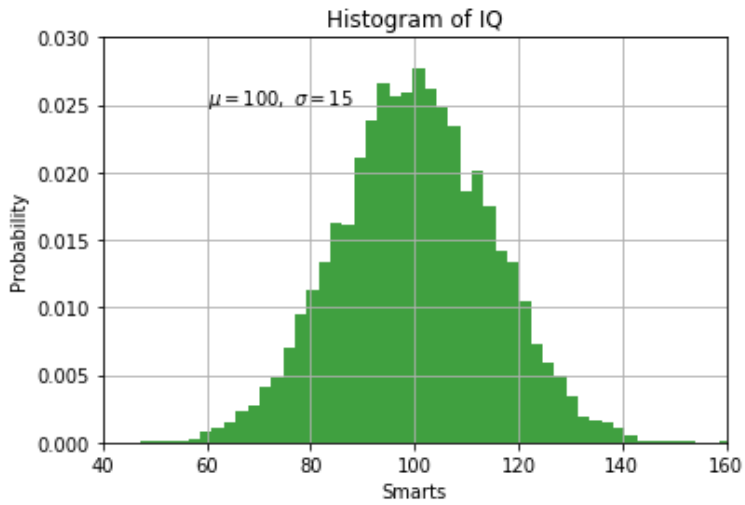


Working with text

- The text() command can be used to add text in an arbitrary location, and the xlabel(), ylabel() and title() are used to add text in the indicated locations.
- Just as with with lines above, you can customize the properties by passing keyword arguments into the text functions or using setp():

  t = plt.xlabel('my data', fontsize=14, color='red')

```
In [25]: mu, sigma = 100, 15
    ...: x = mu + sigma * np.random.randn(10000)
    ...: # the histogram of the data
    ...: n, bins, patches = plt.hist(x, 50, density=1, facecolor='
alpha=0.75)
    ...: plt.xlabel('Smarts')
    ...: plt.ylabel('Probability')
    ...: plt.title('Histogram of IQ')
    ...: plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
    ...: plt.axis([40, 160, 0, 0.03])
    ...: plt.grid(True)
    ...: plt.show()
```



Annotating text

- The uses of the basic text() command above place text at an arbitrary position on the Axes.
- A common use for text is to annotate some feature of the plot, and the annotate() method provides helper functionality to make annotations easy.
- In an annotation, there are two points to consider: the location being annotated represented by the argument xy and the location of the text xytext. Both of these arguments are (x, y) tuples.
- In this basic example, both the xy (arrow tip) and xytext locations (text location) are in data coordinates.

```
In [27]: ax = plt.subplot(111)
    ...:
    ...: t = np.arange(0.0, 5.0, 0.01)
    ...: s = np.cos(2*np.pi*t)
    ...: line, = plt.plot(t, s, lw=2)
    ...:
    ...: plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
    ...:               arrowprops=dict(facecolor='black', shrink=0.
    ...:               )
    ...:
    ...: plt.ylim(-2, 2)
    ...: plt.show()
```