python-tutorial-v4.0

October 1, 2018

1 Python Tutorial

This is a python tutorial built upon **CS231** and **CS228** in Stanford University. We will use the Python programming language for all assignments in this course. We hope you can be more familiar with python after readng this tutorial.

2 Introduction

Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (**numpy**, **scipy**, **matplotlib**, **pandas**, etc.) it becomes a powerful environment for scientific computing.

We expect that many of you will have some experience with Python and numpy; for the rest of you, this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Images
- SciPy: Image Processing, Linear Algebra
- Pandas: DataFrame

2.1 Python versions matter

Python 2.x vs. **Python 3.x**

> Somewhat confusingly, Python 3.x introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.5 and vice versa. For this class all code will use Python 3.x.

Python 2.7 will **NOT** be maintained past 2020. Check the official statement.

> Python 2, thank you for your years of faithful service.
>
> Python 3, **your time is now**.

You can check your Python version at the command line by running:

```
python --version
```

3 Before Getting Started

```
In [1]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

3.1 KISS (Keep It Simple, Stupid)

1. Make your code **accessible** at one's first glance
2. Make your code as **short** as possible

3.2 DRY (Don't Repeat Yourself)

1. Avoid repetition
2. Reuse the module with specific functionality in different places

4 Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

```
In [2]: def quicksort(arr):
            if len(arr) <= 1:
                return arr
            pivot = arr[len(arr) // 2]
```

```
            left = [x for x in arr if x < pivot]
            middle = [x for x in arr if x == pivot]
            right = [x for x in arr if x > pivot]
            return quicksort(left) + middle + quicksort(right)

        print(quicksort([3,6,8,10,1,2,1]))
```

```
[1, 1, 2, 3, 6, 8, 10]
```

4.1 Basic data types

4.1.1 Numbers

Integers and floats work as you would expect from other languages:

```
In [3]: x = 3
        print(x, type(x))
```

```
3 <class 'int'>
```

```
In [4]: print(x + 1)   # Addition;
        print(x - 1)   # Subtraction;
        print(x * 2)   # Multiplication;
        print(x ** 2)  # Exponentiation;
        print(x / 2)
        print(x // 2)
```

```
4
2
6
9
1.5
1
```

```
In [5]: x += 1
        print(x)  # Prints "4"
        x *= 2
        print(x)  # Prints "8"
```

```
4
8
```

```
In [6]: y = 2.5
        print(type(y)) # Prints "<type 'float'>"
        print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x–) operators.

Python also has built-in types for long integers and **complex numbers**; you can find all of the details in the documentation.

4.1.2 Booleans

Python implements all of the usual operators for Boolean logic, but uses **English words** rather than symbols (&&, ||, etc.):

```
In [7]: # ignore irrelevant variables
        t, *_, f = True, False, False, True, False
        print(t)
        print(f)
        print(type(t)) # Prints "<type 'bool'>"

True
False
<class 'bool'>
```

Now we let's look at the operations:

```
In [8]: print(t and f) # Logical AND; t && f
        print(t or f)  # Logical OR; t || f
        print(not t)   # Logical NOT;
        print(t != f)  # Logical XOR;

False
True
False
True
```

```
In [9]: a, b = 1, 2

        a, b = b, a

        print(a)
        print(b)

2
1
```

4.1.3 Strings

```
In [10]: hello = 'hello'   # String literals can use single quotes
         world = "world"   # or double quotes; it does not matter.

         # use single quotation in double quotation, it works vice versa
         s = "I am 'single' quotation"
         print(s)

         print(hello, len(hello))
```

```
I am 'single' quotation
hello 5
```

```
In [11]: hw = hello + ' ' + world  # String concatenation
         print(hw)  # prints "hello world"
```

```
hello world
```

```
In [12]: hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting

         print(hw12)  # prints "hello world 12"

         # you need not to care about parameter types in this manner
         print('{} {} {}'.format(hello, world, 12))

         # this also works since python 3.6
         print(f'{hello} {world} {12}')
```

```
hello world 12
hello world 12
hello world 12
```

String indexing

Forward and Backward

```
In [13]: s = 'hello'
         print(s[0])
         print(s[-1])
         print(s[::-1])
```

```
h
o
olleh
```

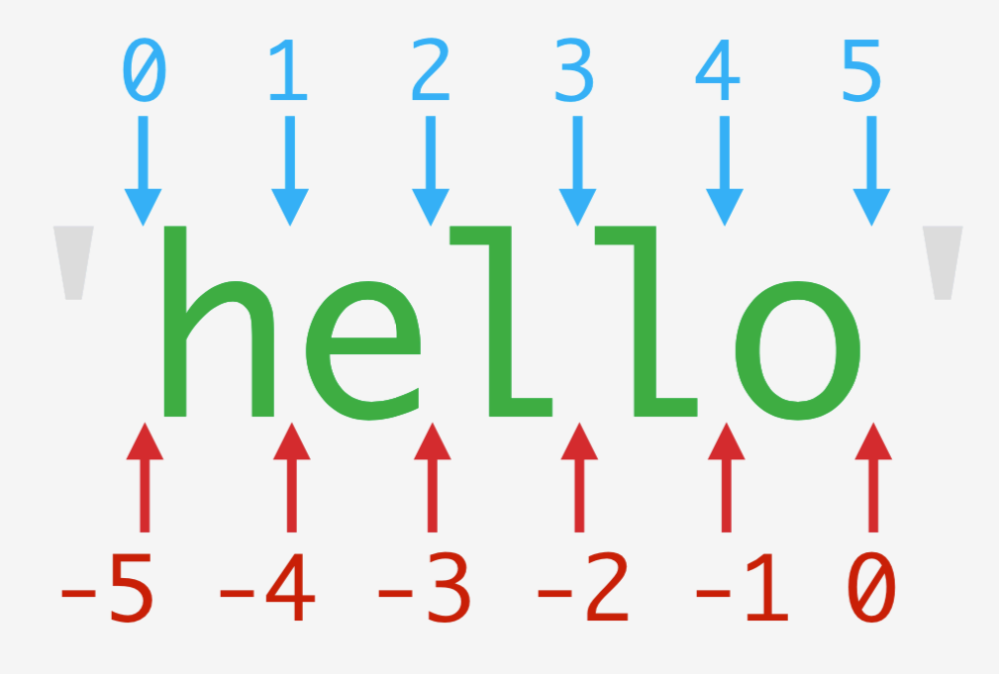


image.png

Manipulate string

```
In [14]: s = 'hello, python!'

         print(s.split(','))

         l = ['hello', ' python!']
         print('-'.join(l))

         print('o' in s)
         print('0' in s)

         # characters in string are immutable
         s[0] = 'H'
```

```
['hello', ' python!']
hello- python!
True
False
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-230010d27143> in <module>()
```

```
        10
        11 # characters in string are immutable
    ---> 12 s[0] = 'H'


        TypeError: 'str' object does not support item assignment
```

String objects have a bunch of useful methods; for example:

```
In [15]: s = "hello"
         print(s.capitalize())  # Capitalize a string; prints "Hello"
         print(s.upper())       # Convert a string to uppercase; prints "HELLO"
         print(s.rjust(7))      # Right-justify a string, padding with spaces; prints "  hello
         print(s.center(7))     # Center a string, padding with spaces; prints " hello "
         print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another
                                         # prints "he(ell)(ell)o"
         print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

You can find a list of all string methods in the documentation.

4.1.4 Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists You can loop over the elements of a list like this:

```
In [16]: animals = ['cat', 'dog', 'monkey']

         for animal in animals:
             print(animal)

         for i in range(len(animals)):
             print(animals[i])
```

```
cat
dog
monkey
cat
dog
monkey
```

List elements can be any type.

```
In [17]: fusion = [[4, 5], 2, 1, 3, 'hello']

         fusion.append('world')
         print(fusion)

         print('hello' in fusion)
         print([5, 4] in fusion)
```

```
[[4, 5], 2, 1, 3, 'hello', 'world']
True
False
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

```
In [18]: animals = ['cat', 'dog', 'monkey']
         for idx, animal in enumerate(animals):
             print('#%d: %s' % (idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

List comprehensions

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
In [19]: nums = [0, 1, 2, 3, 4]
         squares = list()
         for x in nums:
             squares.append(x ** 2)
         print(squares)
```

```
[0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

```
In [20]: nums = [0, 1, 2, 3, 4]
         squares = [x ** 2 for x in nums]
         print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
In [21]: nums = [0, 1, 2, 3, 4]
         even_squares = [x ** 2 for x in nums if x % 2 == 0]
         print(even_squares)
```

```
[0, 4, 16]
```

Dictionaries A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

```
In [22]: d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
         print(d['cat'])       # Get an entry from a dictionary; prints "cute"
         print('cute' in d)     # Check if a dictionary has a given key; prints "True"
```

```
cute
False
```

```
In [23]: d['fish'] = 'wet'    # Set an entry in a dictionary
         print(d['fish'])      # Prints "wet"
```

```
wet
```

```
In [24]: print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

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

<ipython-input-24-78fc9745d9cf> in <module>()
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d


KeyError: 'monkey'
```

```
In [25]: print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
         print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
```

```
N/A
wet
```

```
In [26]: del d['fish']        # Remove an element from a dictionary
         print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

You can find all you need to know about dictionaries in the documentation.
It is easy to iterate over the keys in a dictionary:

```
In [27]: d = {'person': 2, 'cat': 4, 'spider': 8}
         for animal in d:
             legs = d[animal]
             print('A %s has %d legs' % (animal, legs))
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

If you want access to keys and their corresponding values, use the items method:

```
In [28]: d = {'person': 2, 'cat': 4, 'spider': 8}
         for animal, legs in d.items():
             print('A %s has %d legs' % (animal, legs))
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
In [29]: nums = [0, 1, 2, 3, 4]
         even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
         print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

Sets A set is an unordered collection of distinct elements. As a simple example, consider the following:

```
In [30]: animals = {'cat', 'dog'}
         print('cat' in animals)   # Check if an element is in a set; prints "True"
         print('fish' in animals)  # prints "False"
```

```
True
False
```

```
In [31]: animals.add('fish')      # Add an element to a set
         print('fish' in animals)
         print(len(animals))       # Number of elements in a set;
```

```
True
3
```

```
In [32]: animals.add('cat')       # Adding an element that is already in the set does nothing
         print(len(animals))
         animals.remove('cat')    # Remove an element from a set
         print(len(animals))
```

```
3
2
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
In [33]: animals = {'cat', 'dog', 'fish'}
         for idx, animal in enumerate(animals):
             print('#%d: %s' % (idx + 1, animal))
         # Prints "#1: fish", "#2: dog", "#3: cat"
```

```
#1: cat
#2: dog
#3: fish
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
In [34]: from math import sqrt
         print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

Manipulate sets

```
In [35]: st = {1, 2, 3, 4, 5}

         # intersection
         print(st & {1, 3, 5, 7})

         # union
         print(st | {1, 3, 5, 7})

         # diff
         print(st - {1, 3, 5, 7})
```

```
        # union - intersection
        print(st ^ {1, 3, 5, 7})

        # include
        print(st >= {1, 2, 3})
```

```
{1, 3, 5}
{1, 2, 3, 4, 5, 7}
{2, 4}
{2, 4, 7}
True
```

Tuples A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
In [36]: d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
         t = (5, 6)       # Create a tuple
         print(type(t))
         print(d[t])
         print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

```
In [37]: t[0] = 1
```

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

<ipython-input-37-c8aeb8cd20ae> in <module>()
----> 1 t[0] = 1


TypeError: 'tuple' object does not support item assignment
```

4.2 Conditionals

```
In [38]: x = 10

         if 9 <= x <= 11:
             print('Bravo! I could use chanined comparison.')
```

```
Bravo! I could use chanined comparison.
```

4.3 Loops

```
In [39]: import random

         cand = [random.randint(2, 20) for _ in range(10)]

         # pick out prime numbers using for..else..
         for val in cand:
             for i in range(2, val, 1):
                 if val % i == 0:
                     print(f'{val} is not a prime!')
                     break
             else:
                 print(f'{val} is a prime!')
```

```
18 is not a prime!
17 is a prime!
6 is not a prime!
13 is a prime!
14 is not a prime!
17 is a prime!
12 is not a prime!
15 is not a prime!
4 is not a prime!
3 is a prime!
```

4.4 Functions

Python functions are defined using the `def` keyword. For example:

```
In [40]: def sign(x):
             if x > 0:
                 return 'positive'
             elif x < 0:
                 return 'negative'
             else:
                 return 'zero'

         for x in [-1, 0, 1]:
             print(sign(x))
```

```
negative
zero
positive
```

```
In [41]: f = lambda x: x+1
```

```
        list_ = ['1', '2', '3', '4']
        # [1, 2, 3, 4]
        f_list_ = list(map(lambda x: x+'0', list_))
        print(f_list_)

        f(3)
```

```
['10', '20', '30', '40']
```

```
Out[41]: 4
```

We will often define functions to take optional keyword arguments, like this:

```
In [42]: def hello(loud=False, name=3218):
             if loud:
                 print('HELLO, %s' % name.upper())
             else:
                 print('Hello, %s!' % name)

         hello(name='Bob')
         hello(name='Fred', loud=True)
```

```
Hello, Bob!
HELLO, FRED
```

4.4.1 Do your functions have *freestyle*?

```
In [43]: def f(*args, **kwargs):
             for i in args:
                 print(i)
             for key, val in kwargs.items():
                 print(f'{key}: {val}')

         f(1, 2, 4, a=3)

         # this also works
         f(1, 3, 5, 7, you='have', done='a', good='job')
```

```
1
2
4
a: 3
1
3
5
7
you: have
```

```
done: a
good: job
```

4.4.2 Check what you've got in a function

```
In [44]: def func(list_):
             if not isinstance(list_, list):
                 print('This is not list, instead we got {}'.format(type(list_)))
             else:
                 print('Awesome! We got a list.')

         func([1, 2, 3, 4])

         func(1234)
```

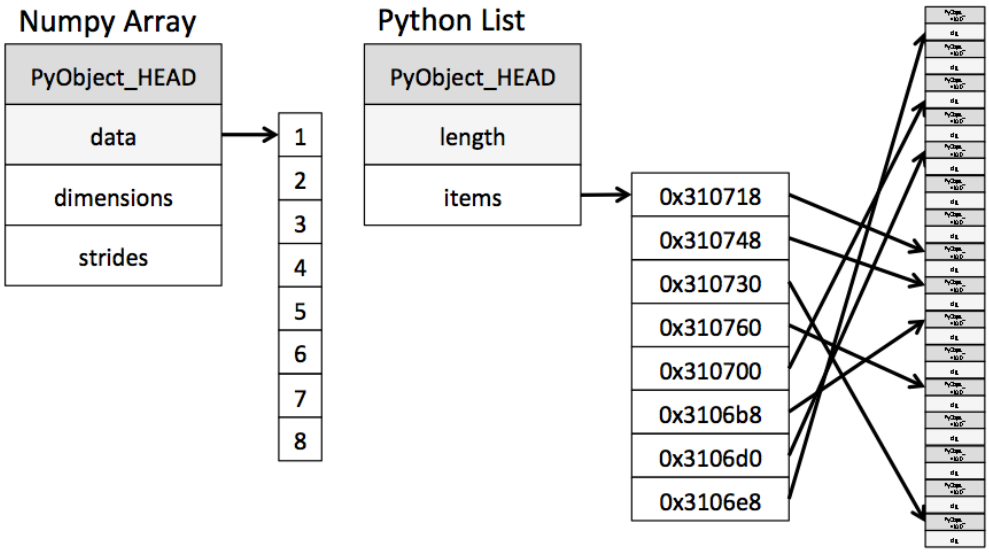
```
Awesome! We got a list.
This is not list, instead we got <class 'int'>
```

4.5 Classes

The syntax for defining classes in Python is straightforward:

```
In [45]: class Greeter:

             # Constructor
             def __init__(self, name):
                 self.name = name  # Create an instance variable
                 self.height = 172

             # Instance method
             def greet(self, loud=False):
                 if loud:
                     print('HELLO, %s!' % self.name.upper())
                 else:
                     print('Hello, %s' % self.name)

         g = Greeter('Fred')  # Construct an instance of the Greeter class
         h = Greeter('Henry')
         h.greet()
         g.greet()            # Call an instance method; prints "Hello, Fred"
         g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

```
Hello, Henry
Hello, Fred
HELLO, FRED!
```



Array vs. List

5 Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

We are going to cover these topics in Numpy:

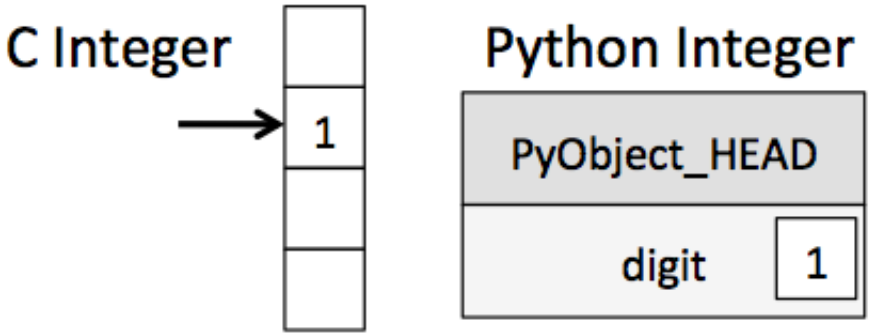
- Arrays
  - Initialization
  - DataType
  - Indexing
  - Mathematics Operation
- Broadcasting

5.0.1 Numpy Arrays vs. Python Lists?

1. Why the need for numpy arrays? Can't we just use Python lists?
2. Iterating over numpy arrays is slow. **Slicing is faster**

From previous slides we know that Python lists may contain items of different types. This flexibility comes at a price: Python lists store *pointers* to memory locations. On the other hand, numpy arrays are typed, where the default type is floating point. Because of this, the system knows how much memory to allocate, and if you ask for an array of size 100, it will allocate one hundred contiguous spots in memory, where the size of each spot is based on the type. This makes access extremely fast.

If you want to know more, we will suggest that you read from Jake Vanderplas's Data Science Handbook. You will find that book an incredible resource for this class.



cint vs. pyint

BUT, **iteration slows things down again**. In general you should not access numpy array elements by iteration. This is because of type conversion. Numpy stores integers and floating points in C-language format. When you operate on array elements through iteration, Python needs to convert that element to a Python int or float, which is a more complex beast (a `struct` in C jargon). This has a cost.

Why is slicing faster? The reason is technical: slicing provides a view onto the memory occupied by a numpy array, instead of creating a new array. That is the reason the code above this cell works nicely as well. However, if you iterate over a slice, then you have gone back to the slow access.

By contrast, functions such as `np.dot` are **implemented at C-level**, do not do this type conversion, and access contiguous memory. If you want this kind of access in Python, use the struct module or Cython. Indeed many fast algorithms in numpy, pandas, and C are either implemented at the C-level, or employ Cython.

To use Numpy, we first need to import the `numpy` package:

```
In [46]: import numpy as np
```

5.1 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
In [47]: a = np.array([1, 2, 3])  # Create a rank 1 array
         print(type(a), a.shape, a[0], a[1], a[2])
         a[0] = 5                 # Change an element of the array
         print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

```
In [48]: b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
         print(b)
```

```
[[1 2 3]
 [4 5 6]]
```

```
In [49]: print(b.shape)
         print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

```
In [50]: a = np.zeros((2,2))  # Create an array of all zeros
         print(a)
```

```
[[0. 0.]
 [0. 0.]]
```

```
In [51]: b = np.ones((1,2))   # Create an array of all ones
         print(b)
```

```
[[1. 1.]]
```

```
In [52]: c = np.full((2,2), 7) # Create a constant array
         print(c)
```

```
[[7 7]
 [7 7]]
```

```
In [53]: d = np.eye(2)        # Create a 2x2 identity matrix
         print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

```
In [54]: e = np.random.random((2,2)) # Create an array filled with random values
         from numpy import random as r
         print(r.random((2, 3)))
         print(e)
```

```
[[0.10815738 0.72765056 0.78283213]
 [0.24448372 0.07773961 0.20663076]]
[[0.95669203 0.94598892]
 [0.50140102 0.81942044]]
```

5.2 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
In [55]: import numpy as np

         # Create the following rank 2 array with shape (3, 4)
         # [[ 1  2  3  4]
         #  [ 5  6  7  8]
         #  [ 9 10 11 12]]
         a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

         # Use slicing to pull out the subarray consisting of the first 2 rows
         # and columns 1 and 2; b is the following array of shape (2, 2):
         # [[2 3]
         #  [6 7]]

         # import copy
         # b = copy.deepcopy(a[:2, 1:-1])
         b = a[:2, 1:-1]
         print(b)
```

```
[[2 3]
 [6 7]]
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

```
In [56]: print(a[0, 1])

         b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
         print(a[0, 1])
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
In [57]: # Create the following rank 2 array with shape (3, 4)
         a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
         print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Two ways of accessing the data in the middle row of the array. Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:

```
In [58]: row_r1 = a[1, :]    # Rank 1 view of the second row of a
         row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
         row_r3 = a[[1], :]  # Rank 2 view of the second row of a
         print(row_r1, row_r1.shape)
         print(row_r2, row_r2.shape)
         print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

```
In [59]: # We can make the same distinction when accessing columns of an array:
         col_r1 = a[:, 1]
         col_r2 = a[:, 1:2]
         print(col_r1, col_r1.shape)
         print
         print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
In [60]: a = np.array([[1,2], [3, 4], [5, 6]])
         print(a.shape)

         print(a)

         # An example of integer array indexing.
         # The returned array will have shape (3,) and
         print(a[[0, 1, 2], [0, 1, 0]])

         # The above example of integer array indexing is equivalent to this:
         print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
(3, 2)
[[1 2]
 [3 4]
 [5 6]]
```

```
[1 4 5]
[1 4 5]
```

```
In [61]: # When using integer array indexing, you can reuse the same
         # element from the source array:
         print(a[[0, 0], [1, 1]])

         # Equivalent to the previous integer array indexing example
         print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
In [62]: # Create a new array from which we will select elements
         a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
         print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
In [63]: # Create an array of indices
         b = np.array([0, 2, 0, 1])

         # Select one element from each row of a using the indices in b
         print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[ 1  6  7 11]
```

```
In [64]: # Mutate one element from each row of a using the indices in b
         a[np.arange(4), b] += 10
         print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
In [65]: import numpy as np

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

         bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                             # this returns a numpy array of Booleans of the same
                             # shape as a, where each slot of bool_idx tells
                             # whether that element of a is > 2.

         print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

```
In [66]: # We use boolean array indexing to construct a rank 1 array
         # consisting of the elements of a corresponding to the True values
         # of bool_idx
         print(a[bool_idx])

         # We can do all of the above in a single concise statement:
         print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing.
In this lecture, we have covered these approaches:

- Number
- Slicing
- Lists
- numpy.array
- boolean array

If you want to know more you should read the documentation.

5.3 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
In [67]: x = np.array([1, 2])  # Let numpy choose the datatype
         y = np.array([1.0, 2.0])  # Let numpy choose the datatype
         z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

         print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

You can read all about numpy datatypes in the documentation.

5.4 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
In [68]: x = np.array([[1,2],[3,4]], dtype=np.float64)
         y = np.array([[5,6],[7,8]], dtype=np.float64)

         # Elementwise sum; both produce the array
         print(x + y)
         print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
In [69]: # Elementwise difference; both produce the array
         print(x - y)
         print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
In [70]: # Elementwise product; both produce the array
         print(x * y)
         print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
In [71]: # Elementwise division; both produce the array
         # [[ 0.2         0.33333333]
         #  [ 0.42857143  0.5       ]]
         print(x / y)
         print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
In [72]: # Elementwise square root; produces the array
         # [[ 1.          1.41421356]
         #  [ 1.73205081  2.        ]]
         print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the **dot** function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
In [73]: x = np.array([[1,2],[3,4]])
         y = np.array([[5,6],[7,8]])

         v = np.array([9, 10])
         w = np.array([11, 12])

         # Inner product of vectors; both produce 219
         print(v.dot(w))
         print(np.dot(v, w))
```

```
219
219
```

```
In [74]: # Matrix / vector product; both produce the rank 1 array [29 67]
         print(x.dot(v))
         print(np.dot(x, v))
```

```
[29 67]
[29 67]
```

```
In [75]: # Matrix / matrix product; both produce the rank 2 array
         # [[19 22]
         #  [43 50]]
         print(x.dot(y))
         print(np.dot(x, y))
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

```
In [76]: x = np.array([[1,2],[3,4]])

         print(np.sum(x))  # Compute sum of all elements; prints "10"
         print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
         print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to **reshape** or otherwise **manipulate** data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

```
In [77]: print(x)
         print(x.T)
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

```
In [78]: v = np.array([[1,2,3]])
         print(v)
         print(v.T)
```

```
[[1 2 3]]
[[1]
 [2]
 [3]]
```

5.5 Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
In [79]: # We will add the vector v to each row of the matrix x,
         # storing the result in the matrix y
         x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
         v = np.array([1, 0, 1])
         y = np.empty_like(x)   # Create an empty matrix with the same shape as x

         # Add the vector v to each row of the matrix x with an explicit loop
         for i in range(4):
             y[i, :] = x[i, :] + v

         print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix x is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv. We could implement this approach like this:

```
In [80]: vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
         print(vv)                  # Prints "[[1 0 1]
                                    #          [1 0 1]
                                    #          [1 0 1]
                                    #          [1 0 1]]"
```

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

```
In [81]: y = x + vv  # Add x and vv elementwise
         print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

```
In [82]: import numpy as np
```

```
        # We will add the vector v to each row of the matrix x,
        # storing the result in the matrix y
        x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
        v = np.array([1, 0, 1])
        y = x + v  # Add v to each row of x using broadcasting
        print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if `v` actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

```
In [83]: # Compute outer product of vectors
         v = np.array([1,2,3])  # v has shape (3,)
         w = np.array([4,5])    # w has shape (2,)
         # To compute an outer product, we first reshape v to be a column
         # vector of shape (3, 1); we can then broadcast it against w to yield
         # an output of shape (3, 2), which is the outer product of v and w:

         print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

```
In [84]: # Add a vector to each row of a matrix
         x = np.array([[1,2,3], [4,5,6]])
         # x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
         # giving the following matrix:

         print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

```
In [85]: # Add a vector to each column of a matrix
         # x has shape (2, 3) and w has shape (2,).
         # If we transpose x then it has shape (3, 2) and can be broadcast
         # against w to yield a result of shape (3, 2); transposing this result
         # yields the final result of shape (2, 3) which is the matrix x with
         # the vector w added to each column. Gives the following matrix:

         print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
In [86]: # Another solution is to reshape w to be a row vector of shape (2, 1);
         # we can then broadcast it directly against x to produce the same
         # output.
         print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
In [87]: # Multiply a matrix by a constant:
         # x has shape (2, 3). Numpy treats scalars as arrays of shape ();
         # these can be broadcast together to shape (2, 3), producing the
         # following array:
         print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

Broadcasting typically makes your code more **concise** and **faster**, so you should strive to use it where possible.

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

6 Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

```
In [88]: import matplotlib.pyplot as plt
```

By running this special iPython command, we will be displaying plots inline in **svg** format:

```
In [89]: %matplotlib inline
         %config InlineBackend.figure_format = 'svg'
```

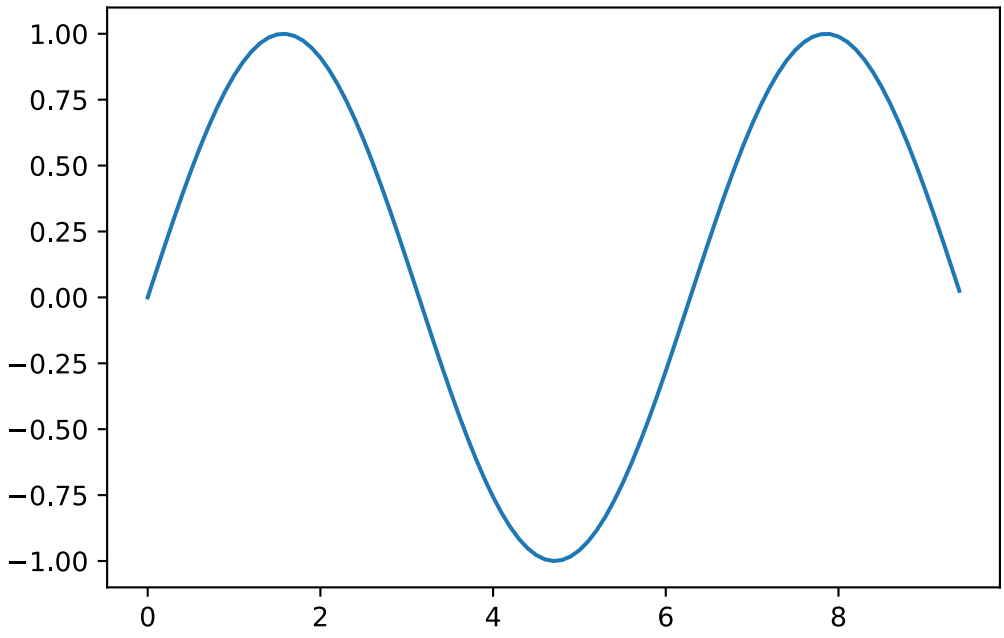
6.1 Plotting

6.1.1 Plot

The **most frequently used** function in `matplotlib` is plot, which allows you to plot 2D data. Here is a simple example:

```
In [90]: # Compute the x and y coordinates for points on a sine curve
         x = np.arange(0, 3 * np.pi, 0.1)
         y = np.sin(x)

         # Plot the points using matplotlib
         plt.plot(x, y)
```

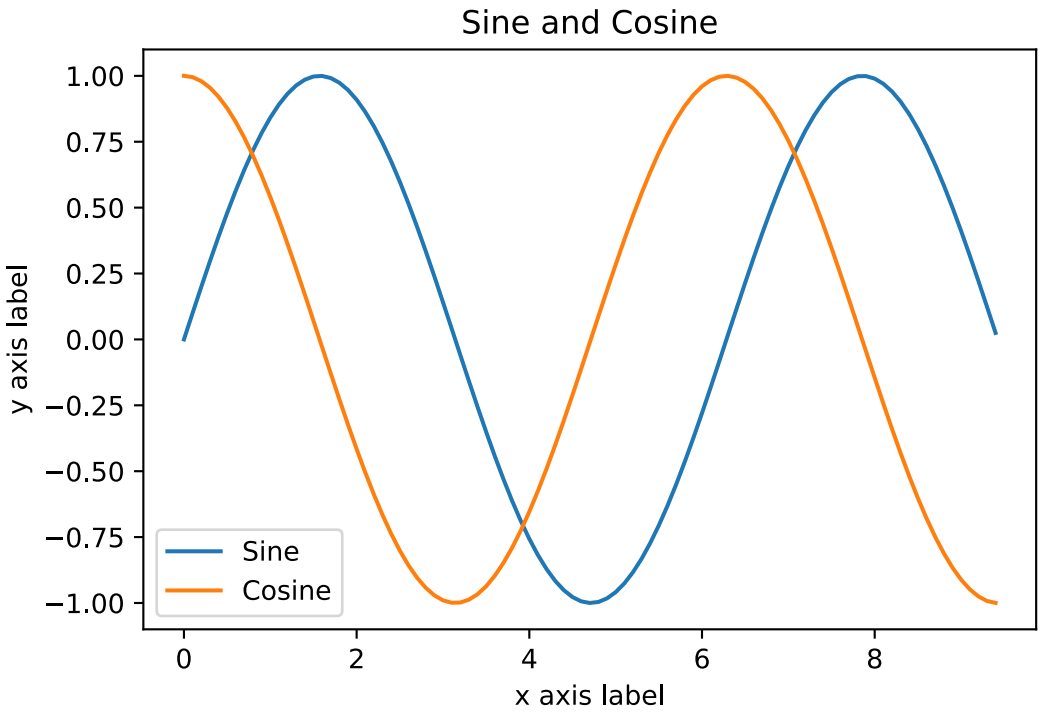
```
Out[90]: [<matplotlib.lines.Line2D at 0x10e37b278>]
```



With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

```
In [91]: y_sin = np.sin(x)
         y_cos = np.cos(x)

         # Plot the points using matplotlib
         plt.plot(x, y_sin)
         plt.plot(x, y_cos)
         plt.xlabel('x axis label')
         plt.ylabel('y axis label')
         plt.title('Sine and Cosine')
         plt.legend(['Sine', 'Cosine'])
```

```
Out[91]: <matplotlib.legend.Legend at 0x11bf4f7f0>
```

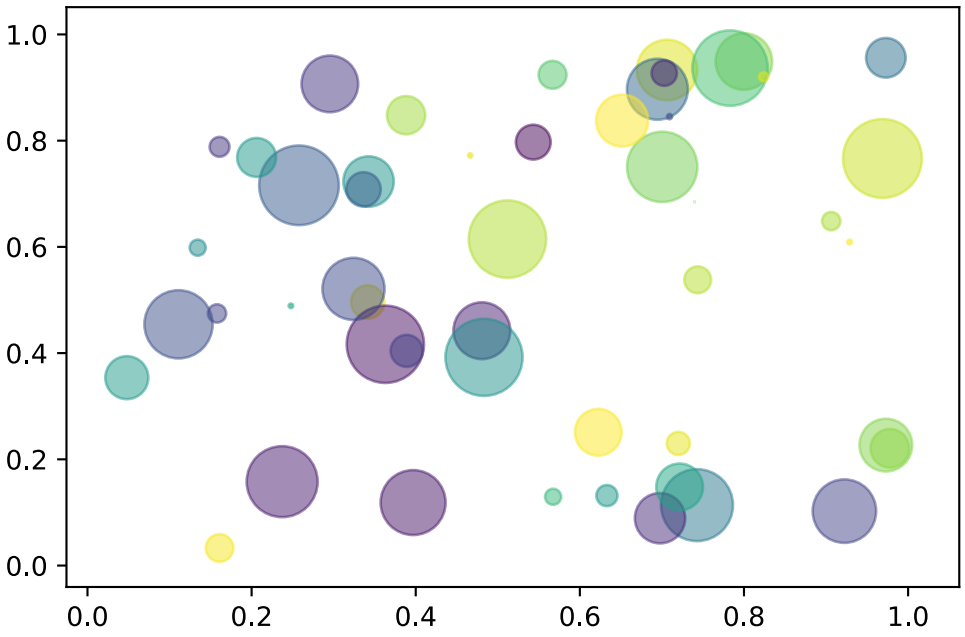


6.1.2 Scatter

BTW, we could plot scatters with adjustable size and transparency.

```
In [92]: np.random.seed(19680801)


         N = 50
         x = np.random.rand(N)
         y = np.random.rand(N)
         colors = np.random.rand(N)
```

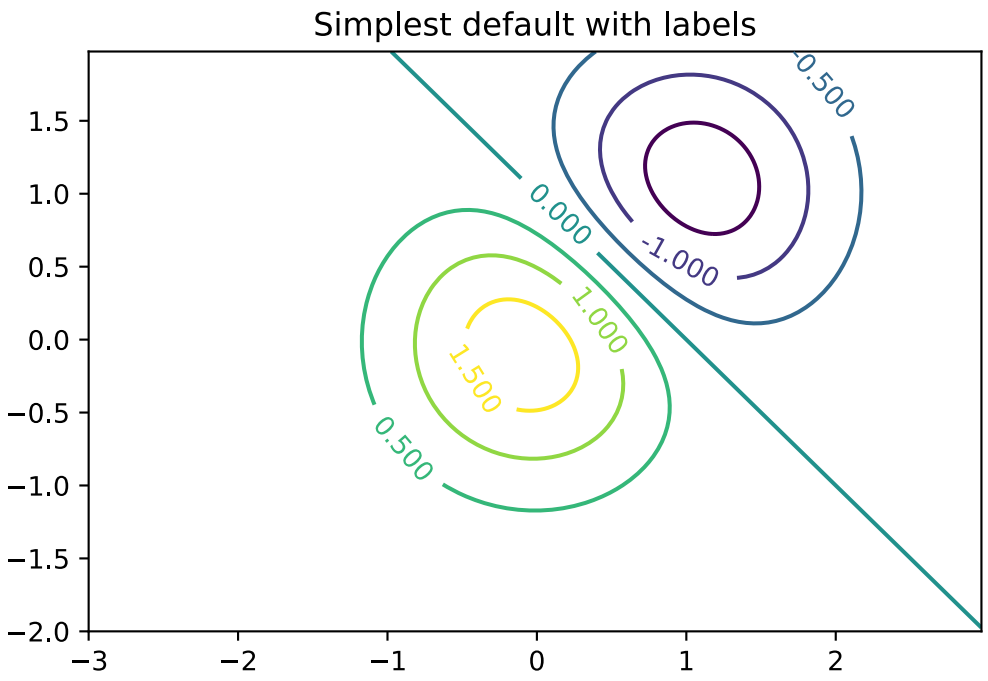
```
area = (30 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



6.1.3 Contour

Draw contour using 3d data

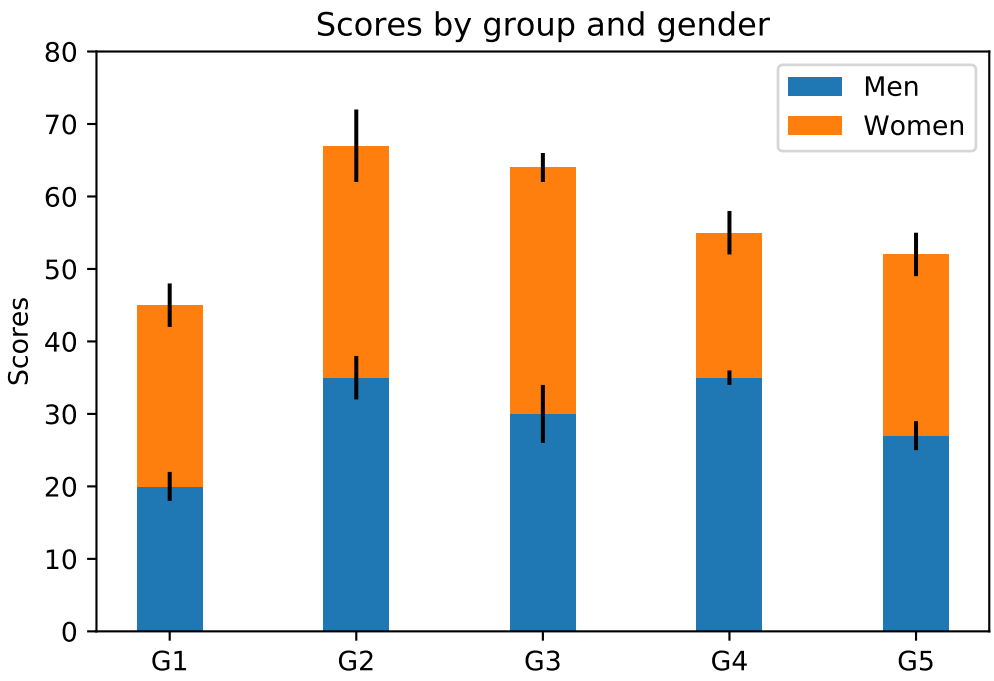
```
In [93]: import matplotlib.cm as cm

         delta = 0.025
         x = np.arange(-3.0, 3.0, delta)
         y = np.arange(-2.0, 2.0, delta)
         X, Y = np.meshgrid(x, y)
         Z1 = np.exp(-X**2 - Y**2)
         Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
         Z = (Z1 - Z2) * 2

         fig, ax = plt.subplots()
         CS = ax.contour(X, Y, Z)
         ax.clabel(CS, inline=1, fontsize=10)
         ax.set_title('Simplest default with labels')

Out[93]: Text(0.5,1,'Simplest default with labels')
```



6.1.4 Bar

Supports bar chart definitely.

```
In [94]: N = 5
         menMeans = (20, 35, 30, 35, 27)
         womenMeans = (25, 32, 34, 20, 25)
         menStd = (2, 3, 4, 1, 2)
         womenStd = (3, 5, 2, 3, 3)
         ind = np.arange(N)    # the x locations for the groups
         width = 0.35       # the width of the bars: can also be len(x) sequence

         p1 = plt.bar(ind, menMeans, width, yerr=menStd)
         p2 = plt.bar(ind, womenMeans, width,
                      bottom=menMeans, yerr=womenStd)

         plt.ylabel('Scores')
         plt.title('Scores by group and gender')
         plt.xticks(ind, ('G1', 'G2', 'G3', 'G4', 'G5'))
         plt.yticks(np.arange(0, 81, 10))
         plt.legend((p1[0], p2[0]), ('Men', 'Women'))

         plt.show()
```



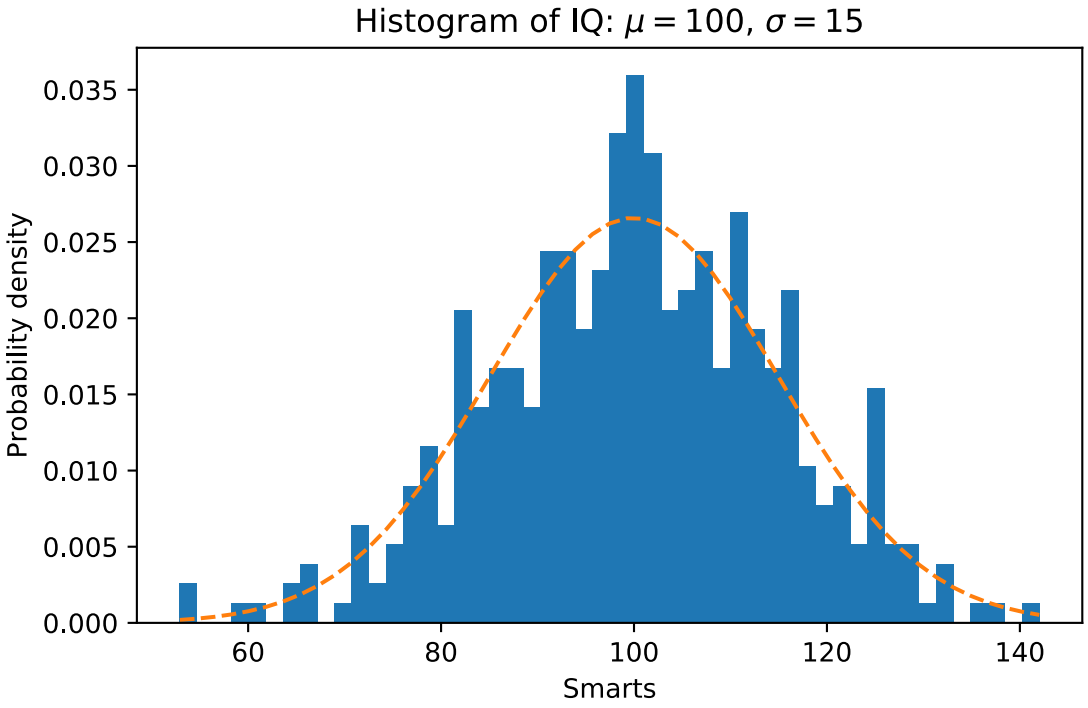
6.1.5 Histogram

How to plot a histogram?

In other words, where should we start to **estimate the distribution** given some data?

```
In [95]: np.random.seed(19680801)

         # example data
         mu = 100  # mean of distribution
         sigma = 15  # standard deviation of distribution
         x = mu + sigma * np.random.randn(437)

         num_bins = 50

         fig, ax = plt.subplots()

         # the histogram of the data
         n, bins, patches = ax.hist(x, num_bins, density=1)

         # add a 'best fit' line
         y = ((1 / (np.sqrt(2 * np.pi) * sigma)) *
              np.exp(-0.5 * (1 / sigma * (bins - mu))**2))
         ax.plot(bins, y, '--')
         ax.set_xlabel('Smarts')
         ax.set_ylabel('Probability density')
```

```
ax.set_title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')

# Tweak spacing to prevent clipping of ylabel
fig.tight_layout()
plt.show()
```

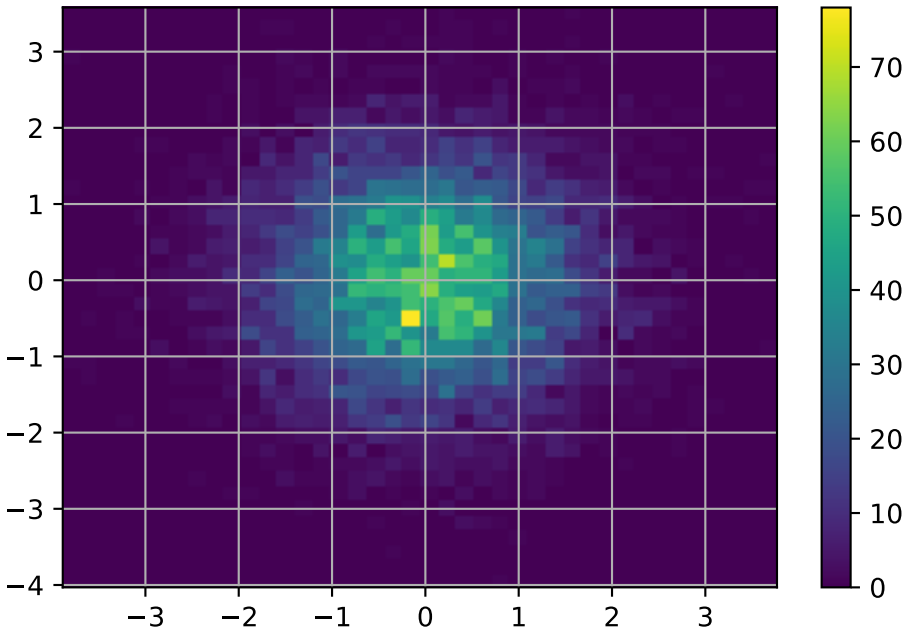


6.1.6 Two-dimensional Histogram

What about 2d histogram?

```
In [96]: mean = [0,0]
         cov = [[0,1],[1,0]]
         x, y = np.random.multivariate_normal(mean, cov, 10000).T
         plt.hist2d(x, y, bins=40)
         plt.colorbar()
         plt.grid()
         plt.show()
```

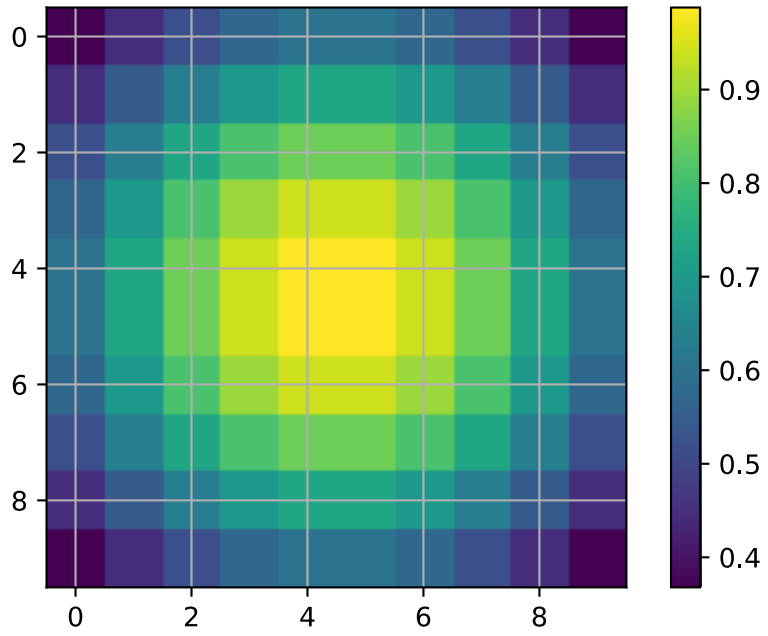
```
/Users/liujintao/Develop/pkg/miniconda3/envs/thu_ids/lib/python3.6/site-packages/ipykernel/__ma
  app.launch_new_instance()
```



6.1.7 "Heatmap"

What about **heatmap**, e.g. a matrix's value?

```
In [97]: # mat = np.random.rand(10, 10)

         # 2d gaussian
         x, y = np.meshgrid(np.linspace(-1,1,10), np.linspace(-1,1,10))
         d = np.sqrt(x*x+y*y)
         sigma, mu = 1.0, 0.0
         mat = np.exp(-( (d-mu)**2 / ( 2.0 * sigma**2 ) ) )

         plt.imshow(mat)
         plt.grid(True)
         plt.colorbar()
         plt.show()
```

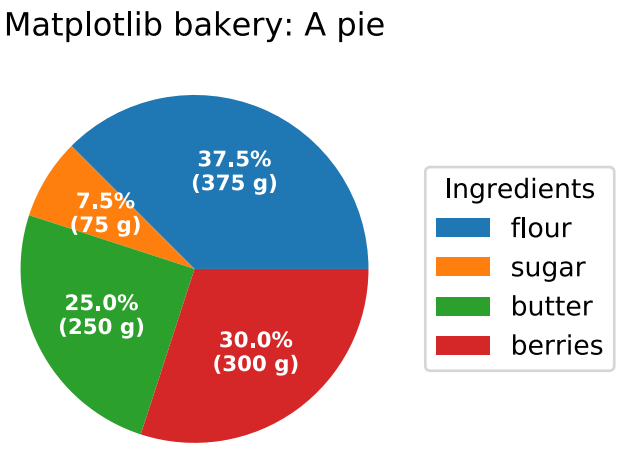


6.1.8 Pie

So does pie chart.

```
In [98]: fig, ax = plt.subplots(figsize=(6, 3), subplot_kw=dict(aspect="equal"))

         recipe = ["375 g flour",
                   "75 g sugar",
                   "250 g butter",
                   "300 g berries"]

         data = [float(x.split()[0]) for x in recipe]
         ingredients = [x.split()[-1] for x in recipe]


         def func(pct, allvals):
             absolute = int(pct/100.*np.sum(allvals))
             return "{:.1f}%\n({:d} g)".format(pct, absolute)


         wedges, texts, autotexts = ax.pie(data, autopct=lambda pct: func(pct, data),
                                           textprops=dict(color="w"))

         ax.legend(wedges, ingredients,
                   title="Ingredients",
                   loc="center left",
```

```
                bbox_to_anchor=(1, 0, 0.5, 1))

plt.setp(autotexts, size=8, weight="bold")

ax.set_title("Matplotlib bakery: A pie")

plt.show()
```



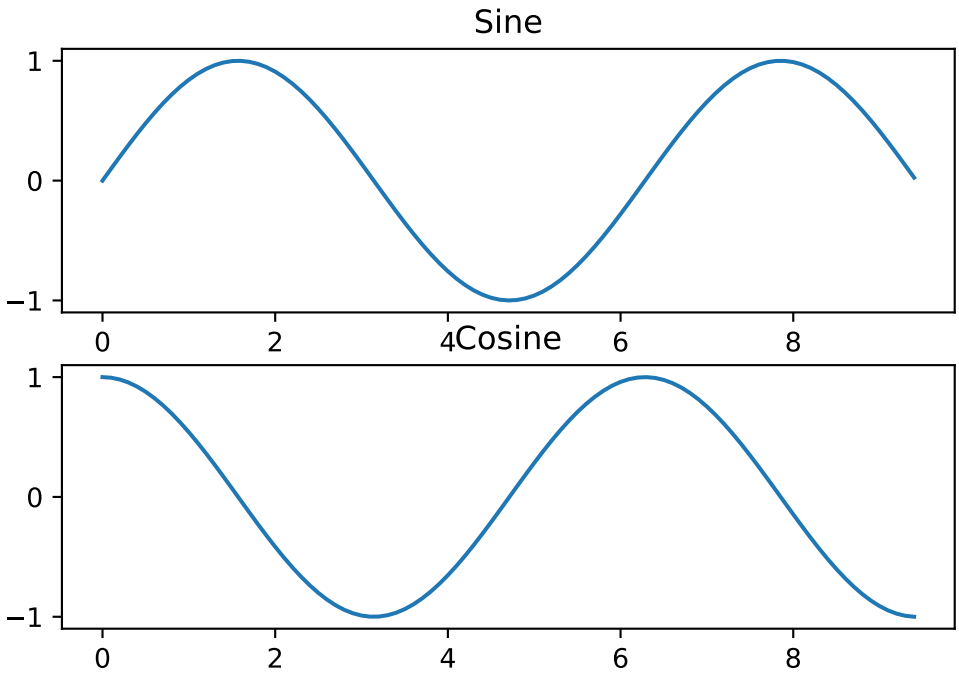
6.2 Subplots

You can read much more about the `subplot` function in the documentation.

You can plot different things in the same figure using the subplot function. Here is an example:

```
In [99]: # Compute the x and y coordinates for points on sine and cosine curves
         x = np.arange(0, 3 * np.pi, 0.1)
         y_sin = np.sin(x)
         y_cos = np.cos(x)

         # Set up a subplot grid that has height 2 and width 1,
         # and set the first such subplot as active.
         plt.subplot(2, 1, 1)

         # Make the first plot
         plt.plot(x, y_sin)
         plt.title('Sine')

         # Set the second subplot as active, and make the second plot.
         plt.subplot(2, 1, 2)
         plt.plot(x, y_cos)
         plt.title('Cosine')
```

```
# Show the figure.
plt.show()
```



6.3 Get tired of memorizing all the APIs?

Check this out!

7 SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

The best way to get familiar with SciPy is to browse the documentation. We will highlight some examples to show the basic operation of SciPy.

7.1 Image operations

SciPy provides some basic functions to work with images. For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images. Here is a simple example that showcases these functions:

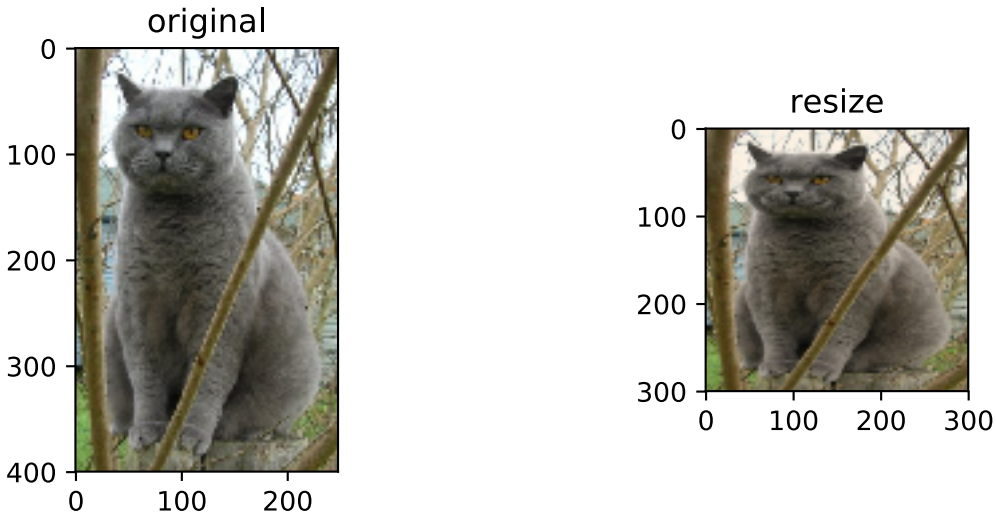
```
In [100]: from PIL import Image
          import numpy as np
```

```
from scipy.ndimage import filters
from matplotlib import pyplot as plt
from scipy.misc import imread, imsave
from skimage.transform import resize
```

```
In [101]: img = np.array(Image.open(u'imgs/cat.jpg').convert('RGB'))


          img_tinted = img * [1, 0.95, 0.9]

          # Resize the tinted image to be 300 by 300 pixels.
          img_tinted = resize(img_tinted, (300, 300), mode='reflect').astype('uint8')

          plt.subplot(1, 3, 1)
          plt.imshow(img)

          plt.title('original')

          plt.subplot(1, 3, 3)
          plt.imshow(img_tinted)
          plt.title('resize')
```

```
/Users/liujintao/Develop/pkg/miniconda3/envs/thu_ids/lib/python3.6/site-packages/skimage/transi
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

```
Out[101]: Text(0.5,1,'resize')
```



7.2 Signal Processing

1. Blurring image

2. Fast Fourier Transform
3. ...

Here is the processing of gaussian filter. It is easy for us to use Scipy package to manipulate different kinds of tools to deal with image.

```
In [102]: im = np.array(Image.open(u'imgs/cat.jpg').convert('L')) #convert to gray picture
          im2 = filters.gaussian_filter(im,3)
          im3 = filters.gaussian_filter(im,5)

          plt.subplot(1,3,1)
          plt.axis('off')
          plt.imshow(im,cmap='gray')
          plt.title('original')

          plt.subplot(1,3,2)
          plt.axis('off')
          plt.imshow(im2,cmap='gray')
          plt.title('gaussian(kernel 3)')

          plt.subplot(1,3,3)
          plt.axis('off')
          plt.imshow(im3,cmap='gray')
          plt.title('gaussian(kernel 5)')

Out[102]: Text(0.5,1,'gaussian(kernel 5)')
```

7.3 Linear Algebra

1. Solving a set of equations

2. Determinants of Square Matrices
3. Inverse of a Square Matrix
4. Singular Value Decomposition
5. ...

```
In [103]: from scipy import linalg
```

7.3.1 Solve functions

Let's suppose this is the set of equations we want to solve:

$$2x + 3y = 7$$
$$3x + 4y = 10$$

```
In [104]: A = np.array([[2, 3], [3, 4]])
          B = np.array([[7], [10]])

          linalg.solve(A, B)
```

```
Out[104]: array([[2.],
                 [1.]])
```

7.3.2 Determinant

To calculate the **determinant** for a square matrix, we can use the `det()` method.

```
In [105]: mat = np.array([[8, 2], [1, 4]])
          linalg.det(mat)
```

```
Out[105]: 30.0
```

7.3.3 Inverse Matrix

We use the `inv()` method to calculate the inverse of a squared matrix.

```
In [106]: linalg.inv(mat)
```

```
Out[106]: array([[ 0.13333333, -0.06666667],
                 [-0.03333333,  0.26666667]])
```

7.3.4 Singular Value Decomposition

In order to perform sigular value decomposition, we simply use `svd()`.

```
In [107]: linalg.svd(mat)
```

```
Out[107]: (array([[-0.9610057 , -0.27652857],
                  [-0.27652857,  0.9610057 ]]),
           array([8.52079729, 3.52079729]),
           array([[-0.9347217 , -0.35538056],
                  [-0.35538056,  0.9347217 ]]))
```

The more detail of SciPy can be found in this page.

8 Pandas

Let's manipulate **dataset** with *pandas*!
So what is the typical form of dataset?

8.1 DataFrame

A **two-dimensional** labeled data structure with columns of **potentially different types**.

- Loading/Creation
- Observation
- Slicing/Indexing
  - Insertion
  - Deletion
  - Update
- Filtering
- Sorting
- ...

8.1.1 Basics

```
In [108]: import pandas as pd

          # Read in the csv files
          dfcars=pd.read_csv("data/mtcars.csv")
          type(dfcars)
```

```
Out[108]: pandas.core.frame.DataFrame
```

```
In [109]: !head -15 data/mtcars.csv
```

```
"","mpg","cyl","disp","hp","drat","wt","qsec","vs","am","gear","carb"
"Mazda RX4",21,6,160,110,3.9,2.62,16.46,0,1,4,4
"Mazda RX4 Wag",21,6,160,110,3.9,2.875,17.02,0,1,4,4
"Datsun 710",22.8,4,108,93,3.85,2.32,18.61,1,1,4,1
"Hornet 4 Drive",21.4,6,258,110,3.08,3.215,19.44,1,0,3,1
"Hornet Sportabout",18.7,8,360,175,3.15,3.44,17.02,0,0,3,2
"Valiant",18.1,6,225,105,2.76,3.46,20.22,1,0,3,1
"Duster 360",14.3,8,360,245,3.21,3.57,15.84,0,0,3,4
"Merc 240D",24.4,4,146.7,62,3.69,3.19,20,1,0,4,2
"Merc 230",22.8,4,140.8,95,3.92,3.15,22.9,1,0,4,2
"Merc 280",19.2,6,167.6,123,3.92,3.44,18.3,1,0,4,4
"Merc 280C",17.8,6,167.6,123,3.92,3.44,18.9,1,0,4,4
"Merc 450SE",16.4,8,275.8,180,3.07,4.07,17.4,0,0,3,3
```

```
"Merc 450SL",17.3,8,275.8,180,3.07,3.73,17.6,0,0,3,3
"Merc 450SLC",15.2,8,275.8,180,3.07,3.78,18,0,0,3,3
```

```
In [110]: dfcars.head(10)
```

Out[110]:

	Unnamed: 0	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
5	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
6	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
7	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
8	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
9	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

The first column is bothersome, how do we clean that up?

```
In [111]: dfcars = dfcars.rename(columns={"Unnamed: 0": "name"})
          dfcars.head()
```

Out[111]:

	name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

To access a column, you can use either **dictionary** syntax or **instance-variable** syntax.

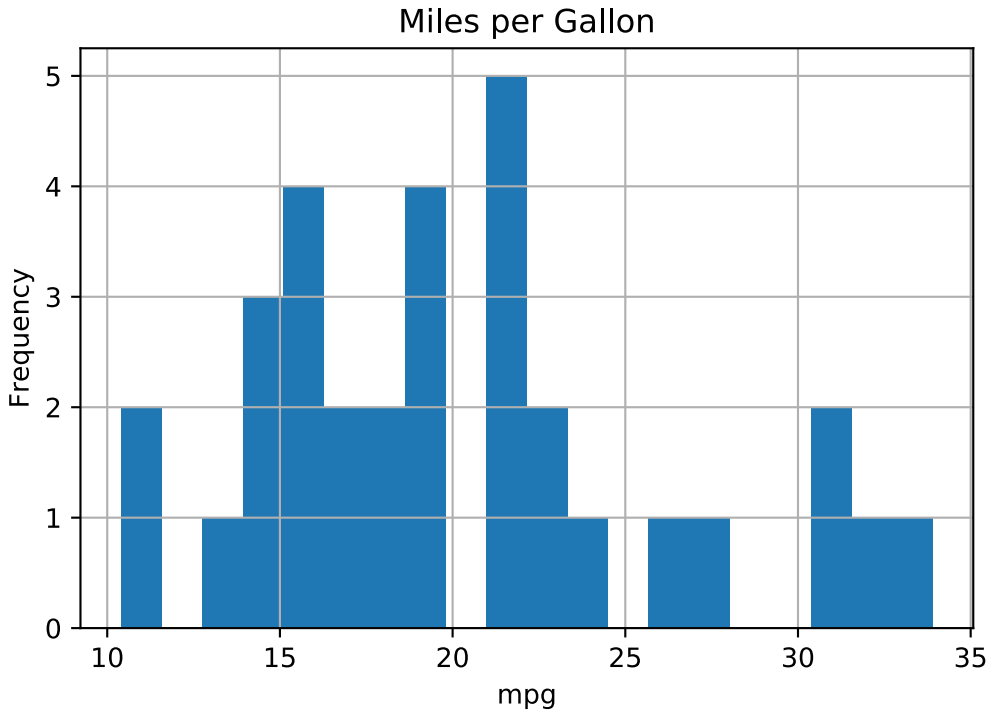
```
In [112]: # print(dfcars.mpg) # This sort of data is called Series.
          # print (dfcars['mpg'])

          print(dfcars.mpg.values) # You can get a numpy array of values frorm the Pandas Seri
          print(type(dfcars.mpg.values))
          # print(type(dfcars.mpg.values))
```

```
[21.  21.  22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.  30.4
 15.8 19.7 15.  21.4]
<class 'numpy.ndarray'>
```

Wanna take a look at the data distribution?

```
In [113]: dfcars.mpg.hist(bins=20)
          # plt.hist(dfcars.mpg.values, bins=20);
          plt.xlabel('mpg')
          plt.ylabel('Frequency')
          plt.title('Miles per Gallon')
```

```
Out[113]: Text(0.5,1,'Miles per Gallon')
```



What if we want to extract a sub-dataframe?

```
In [114]: dfcars[['am', 'mpg']]
```

Out[114]:

	am	mpg
0	1	21.0
1	1	21.0
2	1	22.8
3	0	21.4
4	0	18.7
5	0	18.1
6	0	14.3
7	0	24.4
8	0	22.8
9	0	19.2
10	0	17.8
11	0	16.4
12	0	17.3
13	0	15.2
14	0	10.4
15	0	10.4
16	0	14.7
17	1	32.4
18	1	30.4
19	1	33.9
20	0	21.5
21	0	15.5
22	0	15.2
23	0	13.3
24	0	19.2
25	1	27.3
26	1	26.0
27	1	30.4
28	1	15.8
29	1	19.7
30	1	15.0
31	1	21.4

8.1.2 Descriptive Statistics

Observe the dataset

```
In [115]: print(dfcars.shape) # #items * #attributes

          print(len(dfcars)) # the number of rows in the dataframe

          print(dfcars.columns)
```

```
(32, 12)
32
Index(['name', 'mpg', 'cyl', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs', 'am',
```

```
       'gear', 'carb'],
      dtype='object')
```

```
In [116]: dfcars.info() # including memory usage and counts of null values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 12 columns):
name    32 non-null object
mpg     32 non-null float64
cyl     32 non-null int64
disp    32 non-null float64
hp      32 non-null int64
drat    32 non-null float64
wt      32 non-null float64
qsec    32 non-null float64
vs      32 non-null int64
am      32 non-null int64
gear    32 non-null int64
carb    32 non-null int64
dtypes: float64(5), int64(6), object(1)
memory usage: 3.1+ KB
```

```
In [117]: dfcars.describe() # more statistical values
```

```
Out[117]:              mpg        cyl        disp          hp       drat         wt  \
          count  32.000000  32.000000   32.000000   32.000000  32.000000  32.000000
          mean   20.090625   6.187500  230.721875  146.687500   3.596563   3.217250
          std     6.026948   1.785922  123.938694   68.562868   0.534679   0.978457
          min    10.400000   4.000000   71.100000   52.000000   2.760000   1.513000
          25%    15.425000   4.000000  120.825000   96.500000   3.080000   2.581250
          50%    19.200000   6.000000  196.300000  123.000000   3.695000   3.325000
          75%    22.800000   8.000000  326.000000  180.000000   3.920000   3.610000
          max    33.900000   8.000000  472.000000  335.000000   4.930000   5.424000

                      qsec         vs         am       gear     carb
          count  32.000000  32.000000  32.000000  32.000000  32.0000
          mean   17.848750   0.437500   0.406250   3.687500   2.8125
          std     1.786943   0.504016   0.498991   0.737804   1.6152
          min    14.500000   0.000000   0.000000   3.000000   1.0000
          25%    16.892500   0.000000   0.000000   3.000000   2.0000
          50%    17.710000   0.000000   0.000000   4.000000   2.0000
          75%    18.900000   1.000000   1.000000   4.000000   4.0000
          max    22.900000   1.000000   1.000000   5.000000   8.0000
```

8.1.3 Slice and Filtering

```
In [118]: new_index = np.arange(5, 37)
          dfcars_reindex = dfcars.reindex(new_index)
          dfcars_reindex.head()
```

```
Out[118]:          name   mpg  cyl   disp     hp  drat    wt   qsec   vs   am  gear  \
          5     Valiant  18.1  6.0  225.0  105.0  2.76  3.46  20.22  1.0  0.0   3.0
          6  Duster 360  14.3  8.0  360.0  245.0  3.21  3.57  15.84  0.0  0.0   3.0
          7   Merc 240D  24.4  4.0  146.7   62.0  3.69  3.19  20.00  1.0  0.0   4.0
          8    Merc 230  22.8  4.0  140.8   95.0  3.92  3.15  22.90  1.0  0.0   4.0
          9    Merc 280  19.2  6.0  167.6  123.0  3.92  3.44  18.30  1.0  0.0   4.0

             carb
          5   1.0
          6   4.0
          7   2.0
          8   2.0
          9   4.0
```

We now return the first three rows of dfcars_reindex in two different ways, first with iloc and then with loc.

```
In [119]: dfcars_reindex.iloc[0:3]
```

```
Out[119]:          name   mpg  cyl   disp     hp  drat    wt   qsec   vs   am  gear  \
          5     Valiant  18.1  6.0  225.0  105.0  2.76  3.46  20.22  1.0  0.0   3.0
          6  Duster 360  14.3  8.0  360.0  245.0  3.21  3.57  15.84  0.0  0.0   3.0
          7   Merc 240D  24.4  4.0  146.7   62.0  3.69  3.19  20.00  1.0  0.0   4.0

             carb
          5   1.0
          6   4.0
          7   2.0
```

Since iloc uses the position in the index. Notice that the argument 0:3 with iloc returns the first three rows of the dataframe, which have label names 5, 6, and 7. To access the same rows with loc, we write

```
In [120]: dfcars_reindex.loc[5:7]
```

```
Out[120]:          name   mpg  cyl   disp     hp  drat    wt   qsec   vs   am  gear  \
          5     Valiant  18.1  6.0  225.0  105.0  2.76  3.46  20.22  1.0  0.0   3.0
          6  Duster 360  14.3  8.0  360.0  245.0  3.21  3.57  15.84  0.0  0.0   3.0
          7   Merc 240D  24.4  4.0  146.7   62.0  3.69  3.19  20.00  1.0  0.0   4.0

             carb
          5   1.0
          6   4.0
          7   2.0
```

What if we want to slice both row and column?

```
In [121]: dfcars_reindex.iloc[2:5, 1:4]
          # dfcars_reindex.loc[7:9, 'mpg':'disp']
```

```
Out[121]:    mpg  cyl   disp
          7  24.4  4.0  146.7
          8  22.8  4.0  140.8
          9  19.2  6.0  167.6
```

Usually, we are more intersted in entries meeting some requirements.
Let's do filter by **condition**!

```
In [122]: # dfcars[dfcars['mpg'] >= 20]
          # dfcars[(dfcars['mpg'] >= 20) & (dfcars['hp'] <= 100)]
          dfcars.query("name.str.startswith('T')& hp<=100")
```

```
Out[122]:             name   mpg  cyl   disp  hp  drat     wt   qsec  vs  am  gear  \
          19  Toyota Corolla  33.9    4   71.1  65  4.22  1.835  19.90   1   1     4
          20   Toyota Corona  21.5    4  120.1  97  3.70  2.465  20.01   1   0     3

              carb
          19     1
          20     1
```

8.1.4 Sorting

```
In [123]: # dfcars.sort_values('mpg', ascending=False)
          dfcars.sort_index(axis=1, ascending=True)
```

```
Out[123]:    am  carb  cyl   disp  drat  gear   hp   mpg                name   qsec  \
          0   1     4    6  160.0  3.90     4  110  21.0           Mazda RX4  16.46
          1   1     4    6  160.0  3.90     4  110  21.0       Mazda RX4 Wag  17.02
          2   1     1    4  108.0  3.85     4   93  22.8          Datsun 710  18.61
          3   0     1    6  258.0  3.08     3  110  21.4      Hornet 4 Drive  19.44
          4   0     2    8  360.0  3.15     3  175  18.7   Hornet Sportabout  17.02
          5   0     1    6  225.0  2.76     3  105  18.1             Valiant  20.22
          6   0     4    8  360.0  3.21     3  245  14.3          Duster 360  15.84
          7   0     2    4  146.7  3.69     4   62  24.4           Merc 240D  20.00
          8   0     2    4  140.8  3.92     4   95  22.8           Merc 230  22.90
          9   0     4    6  167.6  3.92     4  123  19.2           Merc 280  18.30
          10  0     4    6  167.6  3.92     4  123  17.8          Merc 280C  18.90
          11  0     3    8  275.8  3.07     3  180  16.4          Merc 450SE  17.40
          12  0     3    8  275.8  3.07     3  180  17.3          Merc 450SL  17.60
          13  0     3    8  275.8  3.07     3  180  15.2         Merc 450SLC  18.00
          14  0     4    8  472.0  2.93     3  205  10.4  Cadillac Fleetwood  17.98
          15  0     4    8  460.0  3.00     3  215  10.4 Lincoln Continental  17.82
          16  0     4    8  440.0  3.23     3  230  14.7   Chrysler Imperial  17.42
          17  1     1    4   78.7  4.08     4   66  32.4            Fiat 128  19.47
```

18	1	2	4	75.7	4.93	4	52	30.4	Honda Civic	18.52
19	1	1	4	71.1	4.22	4	65	33.9	Toyota Corolla	19.90
20	0	1	4	120.1	3.70	3	97	21.5	Toyota Corona	20.01
21	0	2	8	318.0	2.76	3	150	15.5	Dodge Challenger	16.87
22	0	2	8	304.0	3.15	3	150	15.2	AMC Javelin	17.30
23	0	4	8	350.0	3.73	3	245	13.3	Camaro Z28	15.41
24	0	2	8	400.0	3.08	3	175	19.2	Pontiac Firebird	17.05
25	1	1	4	79.0	4.08	4	66	27.3	Fiat X1-9	18.90
26	1	2	4	120.3	4.43	5	91	26.0	Porsche 914-2	16.70
27	1	2	4	95.1	3.77	5	113	30.4	Lotus Europa	16.90
28	1	4	8	351.0	4.22	5	264	15.8	Ford Pantera L	14.50
29	1	6	6	145.0	3.62	5	175	19.7	Ferrari Dino	15.50
30	1	8	8	301.0	3.54	5	335	15.0	Maserati Bora	14.60
31	1	2	4	121.0	4.11	4	109	21.4	Volvo 142E	18.60

	vs	wt
0	0	2.620
1	0	2.875
2	1	2.320
3	1	3.215
4	0	3.440
5	1	3.460
6	0	3.570
7	1	3.190
8	1	3.150
9	1	3.440
10	1	3.440
11	0	4.070
12	0	3.730
13	0	3.780
14	0	5.250
15	0	5.424
16	0	5.345
17	1	2.200
18	1	1.615
19	1	1.835
20	1	2.465
21	0	3.520
22	0	3.435
23	0	3.840
24	0	3.845
25	1	1.935
26	0	2.140
27	1	1.513
28	0	3.170
29	0	2.770
30	0	3.570
31	1	2.780

8.1.5 Update

- Update a grid
- Update a column
- ...

```
In [124]: dfcars.head()
          dfcars.iloc[0, 0] = 'Good Car'
          dfcars.head()
```

```
Out[124]:                 name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  \
          0           Good Car  21.0    6  160.0  110  3.90  2.620  16.46   0   1     4
          1      Mazda RX4 Wag  21.0    6  160.0  110  3.90  2.875  17.02   0   1     4
          2         Datsun 710  22.8    4  108.0   93  3.85  2.320  18.61   1   1     4
          3     Hornet 4 Drive  21.4    6  258.0  110  3.08  3.215  19.44   1   0     3
          4  Hornet Sportabout  18.7    8  360.0  175  3.15  3.440  17.02   0   0     3

             carb
          0     4
          1     4
          2     1
          3     1
          4     2
```

```
In [125]: print(dfcars.head())
          dfcars.cyl -= 1
          dfcars.head()
```

```
                name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  \
0           Good Car  21.0    6  160.0  110  3.90  2.620  16.46   0   1     4
1      Mazda RX4 Wag  21.0    6  160.0  110  3.90  2.875  17.02   0   1     4
2         Datsun 710  22.8    4  108.0   93  3.85  2.320  18.61   1   1     4
3     Hornet 4 Drive  21.4    6  258.0  110  3.08  3.215  19.44   1   0     3
4  Hornet Sportabout  18.7    8  360.0  175  3.15  3.440  17.02   0   0     3

   carb
0     4
1     4
2     1
3     1
4     2
```

```
Out[125]:                 name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  \
          0           Good Car  21.0    5  160.0  110  3.90  2.620  16.46   0   1     4
          1      Mazda RX4 Wag  21.0    5  160.0  110  3.90  2.875  17.02   0   1     4
          2         Datsun 710  22.8    3  108.0   93  3.85  2.320  18.61   1   1     4
          3     Hornet 4 Drive  21.4    5  258.0  110  3.08  3.215  19.44   1   0     3
          4  Hornet Sportabout  18.7    7  360.0  175  3.15  3.440  17.02   0   0     3
```

```
   carb
0     4
1     4
2     1
3     1
4     2
```

8.1.6 Insertion and Deletion

```
In [126]: dfcars['like'] = 'yes'
          dfcars.head()
```

Out[126]:

	name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	like
0	Good Car	21.0	5	160.0	110	3.90	2.620	16.46	0	1	4	4	yes
1	Mazda RX4 Wag	21.0	5	160.0	110	3.90	2.875	17.02	0	1	4	4	yes
2	Datsun 710	22.8	3	108.0	93	3.85	2.320	18.61	1	1	4	1	yes
3	Hornet 4 Drive	21.4	5	258.0	110	3.08	3.215	19.44	1	0	3	1	yes
4	Hornet Sportabout	18.7	7	360.0	175	3.15	3.440	17.02	0	0	3	2	yes

```
In [127]: dfcars.drop(columns=['disp', 'hp']) # this does not modify DataFrame inplace
```

Out[127]:

	name	mpg	cyl	drat	wt	qsec	vs	am	gear	carb	like
0	Good Car	21.0	5	3.90	2.620	16.46	0	1	4	4	yes
1	Mazda RX4 Wag	21.0	5	3.90	2.875	17.02	0	1	4	4	yes
2	Datsun 710	22.8	3	3.85	2.320	18.61	1	1	4	1	yes
3	Hornet 4 Drive	21.4	5	3.08	3.215	19.44	1	0	3	1	yes
4	Hornet Sportabout	18.7	7	3.15	3.440	17.02	0	0	3	2	yes
5	Valiant	18.1	5	2.76	3.460	20.22	1	0	3	1	yes
6	Duster 360	14.3	7	3.21	3.570	15.84	0	0	3	4	yes
7	Merc 240D	24.4	3	3.69	3.190	20.00	1	0	4	2	yes
8	Merc 230	22.8	3	3.92	3.150	22.90	1	0	4	2	yes
9	Merc 280	19.2	5	3.92	3.440	18.30	1	0	4	4	yes
10	Merc 280C	17.8	5	3.92	3.440	18.90	1	0	4	4	yes
11	Merc 450SE	16.4	7	3.07	4.070	17.40	0	0	3	3	yes
12	Merc 450SL	17.3	7	3.07	3.730	17.60	0	0	3	3	yes
13	Merc 450SLC	15.2	7	3.07	3.780	18.00	0	0	3	3	yes
14	Cadillac Fleetwood	10.4	7	2.93	5.250	17.98	0	0	3	4	yes
15	Lincoln Continental	10.4	7	3.00	5.424	17.82	0	0	3	4	yes
16	Chrysler Imperial	14.7	7	3.23	5.345	17.42	0	0	3	4	yes
17	Fiat 128	32.4	3	4.08	2.200	19.47	1	1	4	1	yes
18	Honda Civic	30.4	3	4.93	1.615	18.52	1	1	4	2	yes

```
19       Toyota Corolla  33.9    3  4.22  1.835  19.90    1    1     4     1
20        Toyota Corona  21.5    3  3.70  2.465  20.01    1    0     3     1
21     Dodge Challenger  15.5    7  2.76  3.520  16.87    0    0     3     2
22          AMC Javelin  15.2    7  3.15  3.435  17.30    0    0     3     2
23           Camaro Z28  13.3    7  3.73  3.840  15.41    0    0     3     4
24     Pontiac Firebird  19.2    7  3.08  3.845  17.05    0    0     3     2
25            Fiat X1-9  27.3    3  4.08  1.935  18.90    1    1     4     1
26        Porsche 914-2  26.0    3  4.43  2.140  16.70    0    1     5     2
27         Lotus Europa  30.4    3  3.77  1.513  16.90    1    1     5     2
28       Ford Pantera L  15.8    7  4.22  3.170  14.50    0    1     5     4
29         Ferrari Dino  19.7    5  3.62  2.770  15.50    0    1     5     6
30        Maserati Bora  15.0    7  3.54  3.570  14.60    0    1     5     8
31           Volvo 142E  21.4    3  4.11  2.780  18.60    1    1     4     2
```

```
   like
0   yes
1   yes
2   yes
3   yes
4   yes
5   yes
6   yes
7   yes
8   yes
9   yes
10  yes
11  yes
12  yes
13  yes
14  yes
15  yes
16  yes
17  yes
18  yes
19  yes
20  yes
21  yes
22  yes
23  yes
24  yes
25  yes
26  yes
27  yes
28  yes
29  yes
30  yes
31  yes
```

```
In [128]: dfcars.loc['inserted'] = ('My Car', 22., 1, 329.0, 10., 3.2, 2.3, 21.2, 0, 1, 3, 1,
          dfcars.tail()
```

Out[128]:

	name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	like
28	Ford Pantera L	15.8	7	351.0	264.0	4.22	3.17	14.5	0	1	5	4	yes
29	Ferrari Dino	19.7	5	145.0	175.0	3.62	2.77	15.5	0	1	5	6	yes
30	Maserati Bora	15.0	7	301.0	335.0	3.54	3.57	14.6	0	1	5	8	yes
31	Volvo 142E	21.4	3	121.0	109.0	4.11	2.78	18.6	1	1	4	2	yes
inserted	My Car	22.0	1	329.0	10.0	3.20	2.30	21.2	0	1	3	1	no

```
In [129]: dfcars.drop(index='inserted', inplace=True) # the line added just now is deleted thi
          dfcars.tail()
```

Out[129]:

	name	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	like
27	Lotus Europa	30.4	3	95.1	113.0	3.77	1.513	16.9	1	1	5	2	yes
28	Ford Pantera L	15.8	7	351.0	264.0	4.22	3.170	14.5	0	1	5	4	yes
29	Ferrari Dino	19.7	5	145.0	175.0	3.62	2.770	15.5	0	1	5	6	yes
30	Maserati Bora	15.0	7	301.0	335.0	3.54	3.570	14.6	0	1	5	8	yes
31	Volvo 142E	21.4	3	121.0	109.0	4.11	2.780	18.6	1	1	4	2	yes

9 References

Python Basics

- Python tutorial in Chinese by Xuefeng Liao()
- Style guides for Google-originated open-source projects

Modules

- Numpy User Guide
- Matplotlib Overview
- Scipy User Guide

- Pandas Documentation
- Jupyter Notebook Tutorial

Courses

- Foundations of Data Science@Berkeley
- Python for Data Science@Berkeley

Booklets

- Cheatsheet for Data Science
- Sample plots in Matplotlib
- Data Fair
  - This is a brief but delightful tutorial covering python, big data systems, ...
- STA-663-2017
  - This is a rather shorter tutorial, mainly focused on **statistical** related tools

Books for later study

- Effective Python