CHALMERS UNIVERSITY OF TECHNOLOGY
Fluid Dynamics, M2
412 96 Göteborg

MTF072 Computational Fluid Dynamics

A.S. Kannan ,G. Montero Villar, H. Nilsson November 3, 2019

Introduction to Python

Python is a widely used general-purpose, high level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for improving the emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code. Like *MATLAB*, Python is also an interpreted language. This would mean that Python code can be ported between all of the major operating system platforms and CPU architectures out there, with only minor changes required for each individual platform.

Why python?

Python is widely regarded as the most popular language for *scientific computing*. There has been a steady shift towards python over the past decade, with an increasing demand for skilled python developers in both industry and academia. This is because there are a plethora of possibilities with it, a few of which are listed below -

- Web development – Web frameworks like Django and Flask are based on Python. They help you write server side code which helps you manage databases, write back-end programming logic, mapping urls etc.
- Machine learning – There are several modules available in Python to help programmers implement machine learning tasks for e.g. NumPy, SciPy and scikit-learn modules.
- Data Analysis – Advanced data analytics and visualization packages are available in Python to aid in state-of-the-art data handling and post-processing (e.g. NumPy, SciPy, scikit-learn, matplotlib, pandas and several more)
- Scripting and automation – Python can be used to automate several mundane tasks such as sending automated responses to emails, scheduling automatic backups, tracking server/work station performance, running and monitoring several instances of an application (for e.g. a simulation) etc.

We would like to offer the possibility for students to get familiar with python over the course of the computer tasks done in this course. The following document aims to provide some basic information on setting up a Python environment (in relation to the computer assignments). A cheat sheet with the most commonly used commands is attached at the end of this document (to kick start your python journey).

Installation and basic setup

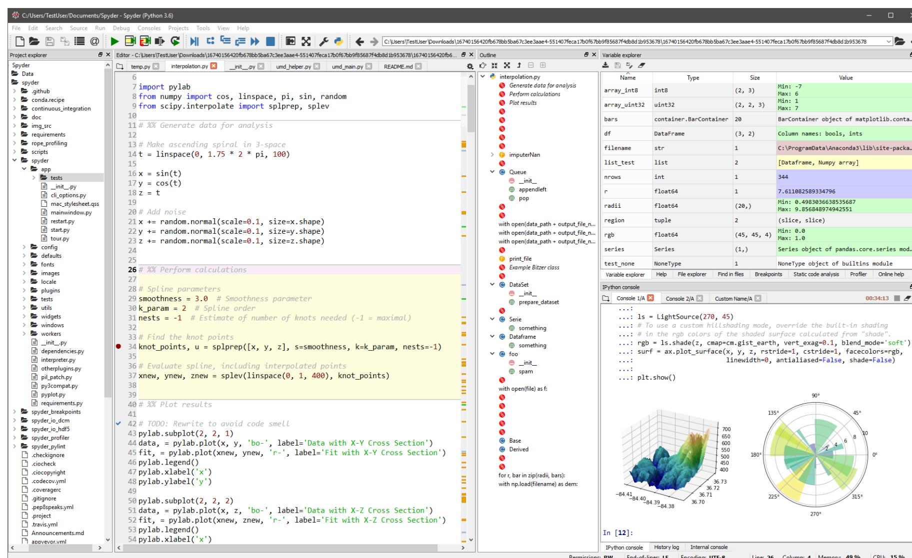
For the purposes of creating an environment that is very similar to *MATLAB*, and ensuring a comprehensive installation of all the relevant packages as well as the latest versions of Python we would recommend using the python distribution manager : Anaconda; which is a package manager, an environment manager, a Python/R data science distribution, and a collection of over 1,500+ open source packages. As of this writing, there are two major versions of Python available: Python 2 and Python 3. You should definitely install the version of Anaconda for Python 3, since Python 2 will not be supported past January 1, 2020. Refer to the installation instructions below to setup the Python environment in your machine. For those of you who are already familiar with python2 and would like to move to python3, this link link is a useful resource explaining all the new features.

- Installation in Windows: Download the anaconda windows installer and follow the instructions listed here.
- Installation in MacOS: Download the anaconda macOS installer and follow the instructions listed here.
- Installation in Linux: Follow the instructions listed here.

Spyder: The Scientific Python Development Environment

Spyder is a powerful scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. It offers a unique combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package. The easiest way to get up and running with Spyder is to download it as part of the Anaconda distribution, and use the conda package and environment manager to keep it and your other packages installed and up to date. We recommend the latest 64-bit Python 3 version, unless you have specific requirements that dictate otherwise.

Setting up a python project using the Spyder IDE would enable improved functionality. Moreover, syntactic errors are easily traced and auto-complete suggestions easily accessible. Spyder provides a *MATLAB* like interface which can be used to monitor stored data-arrays and other program specific datasets. Refer to this detailed tutorial explaining all the functionalities within Spyder for a more detailed insight.



Spyder: The Scientific Python Development Environment

Basics in python programming: Hello world!

Python is a general purpose, high-level, object-oriented language. It's an *interpreted* language *i.e* the code is translated into machine code at runtime. **An important feature of the Python language is the use of indentation for code blocking** instead of braces "{ }". Statements are terminated with a new line instead of ";" although you can use a back-slash "\" to split up a long statement over several lines for readability.

```

a = 0
while a != 5:
      print(a)              # This is an indented..
      a += 1                # ..block of code



while a != 5:
print(a)                    # This would give an indentation error
a += 1

a = 100 + \
    200 + \
    300                     # Breaking up a statement over many lines
```

Listing 1: Python Example

Data handling

Identifiers are the names assigned to variables, classes, functions and objects among others. They cannot start with a number or contain special

characters except for “_” and are case sensitive. Class names usually start with an upper-case letter. When using variables, you need not explicitly declare the type of data it contains. The standard data-types in python include *numbers*, *strings*, *lists*, *tuples*, and *dictionaries*.

```
1 i = 10
2 Count = 100
3 T = 273.15  # Kelvin
4 P = 101325.00     # Pa
5 Data_file = "react.xml" # A string
6 Z = 2 + 3j  # A complex number
7
8 # multiple assignments on the same line
9 T,P,Data_file = 273.15, 101325, "react.xml"
```

Lists are versatile and can be quite useful for your code. The elements of a list are enclosed using square brackets “[ ]” and separated by a comma. The elements need not be of the same type and can also be lists themselves. *NumPy arrays* are likely more useful for scientific calculation, they contain elements of the same type (say a floating point number) and can be manipulated in ways similar to lists using some *NumPy* specific commands.

```
>>> list1 = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
>>> list2 = [123, 'john']
>>> print(list1)        # Prints complete list
['abcd', 786, 2.23, 'john', 70.2]
>>> print(list1[0])     # Prints first element of the list
abcd
>>> print(list1[1:3])   # Prints elements starting from 2nd till 3rd
[786, 2.23]
>>> print(list1[2:])    # Prints elements starting from 3rd element
[2.23, 'john', 70.2]
>>> print(list2 * 2)    # Prints list two times
[123, 'john', 123, 'john']
>>> print(list1 + list2)      # Prints concatenated lists
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
>>> len(list)           # returns length of a list
5
>>> print(list1[-1])    # reverse counting
70.2
>>> del list1[2]  # delete a list element
>>> print(list1)
['abcd', 786, 'john', 70.2]
>>> flag = 786 in list1 # keyword 'in' to check membership
>>> print(flag)
True
>>> print 2.23 in list1
False
>>> list1.append('new')
>>> print(list1)
['abcd', 786, 'john', 70.2, 'new']
>>> list1[1], list1[3] = 900, 'Mary'
>>> print(list1)
['abcd', 900, 'john', 'Mary', 'new']
>>> list1[1], list1[3] = 900, 'Mary'
>>> print(list1)
['abcd', 900, 'john', 'Mary', 'new']
```

There are keywords for other list operations such as sorting, reversing, maximum/minimum and so on.

Conditionals and loops

There are several useful ways in which you can control the execution of your code. Listed below are some e.gs for conditionals and loops.

```
1  var = 100
2  if var < 200:             # Do not forget the ':'
3     print("Expression value is less than 200")
4     if var == 150:
5        print("Which is 150")    # notice the indentation
6     elif var == 100:
7        print("Which is 100")
8     elif var == 50:
9        print("Which is 50")
10    elif var < 50:
11       print("Expression value is less than 50")
12 else:
13    print("Could not find true expression")
```

Listing 2: Some nested 'if' statements

```
Expression value is less than 200
Which is 100
```

```
1  '''
2  Several options for 'for' loops. The syntax is--
3
4  for <iterator> in <sequence>:
5                  statement 1
6                  statement 2
7                  ...
8  '''
9  for i in "some string abcd":
10         print(i)
11
12 T = [100.0, 200.0, 300.0, 'abc']
13
14 for i2 in T:
15         print(i2)
16
17 for i in range(1,20,3): # range() produces a sequence of integers
18         print(i)
19
20 for i in range(0,len(T)): # iterate by sequence
21         print(T[i])
```

Listing 3: 'for' loops

Functions

User defined functions are a convenient way to perform some often repeated computation. You can pass one or more arguments to a function. A function

must be defined in your code before it can be called. You can call a function from the Python prompt as well, once it has been defined or your script has been run. **NOTE: When you pass a variable into a function when calling it, it is always passed by *reference*, *i.e* the function will not create it's own working copy but will work on the memory location which contains the variable**. Variables declared inside the function only have local scope.

```
'''

def functionname( parameters ):
      "Some description aka doc_string"
      statement 1
      statement 2
      ...
      ..
      return [expression]
'''
def my_func(a, b, c = 10):      # a `default' value for c
        "Prints a string, adds three numbers"

        str1 = "some other data"
        print(str1)      # try: comment out the previous line
        return a + b + c


str1 = "some data "
print("add(2,3) : ", my_func(2,3))
print("add(2,3,5) : ", my_func(2,3,5))
print(str1)


def my_func2(listx):
        t = [1, 'abc']
        listx.append(t)
        return

list1 = [300, 100.5]
print("list 1 = ", list1)
my_func2(list1)          # argument pass by reference
print("list 1 is now", list1)
```

NumPy

NumPy is a python package for scientific computing. It can be used to lend some *MATLAB*-like features to your code.

```
import numpy as np      # import the package and give it a convenient
    alias eg `np'

a = np.linspace(1,10,10)      # 10 linearly spaced points from 1 to 10
print("a=", a)

b = np.logspace(1,10,5)       # 5 points on a log scale
print("b=", b)

print("sqrt(5) = " , np.sqrt(5))
```

```
print("5^(2/3) = " , np.power(5.00,2.00/3.00))
print("sqrt(a) = " , np.sqrt(a))

c = np.append(a,b)
print("c=", c)

# note (3,3) is a 'tuple', which is standard datatype. Tuples are
    immutable
z = np.zeros((3,3))
print("z=" ,z)

A = np.array([100.0, 200.0, 300.0, 400.0, 500.0])
print("A[2:]=" , A[2:])          # manipulate the array like you would
    a list using ':'
print("max(A)==" , np.max(A))
```

For more examples, see NumPy Tutorials.

Plotting results

The *matplotlib* package can be used to plot your results.

```
import matplotlib.pyplot as plt

x = np.linspace(0,99,1000)
y = np.power(x, 1.0/3.0)


fig1 = plt.figure("Figure 1")
plt.plot(x,y,'g-',x,y2,'r--')
plt.xlabel("x [units]")
plt.ylabel("y [units]")
plt.legend((" $ x = y^{1/3} $","$ y = sin(x) $" ))  # latex like
    expressions inside $..$
plt.title("Title")
fig1.show()
fig1.savefig("my_fig")
```

For more examples, see matplotlib tutorials.

Appendix: Some python cheat sheets

In this section you can find attached some commonly available cheat sheets. Note that these are only provided here to help you get started with python. **The most comprehensive bible for python and py related help is of course *Google*, so don't hesitate to google your py related issues/-queries...**

PYTHON

Cheat Sheet

codewithmosh.com

Variables

```
a = 1 (integer)
b = 1.1 (float)
c = 1 + 2j (complex)
d = “a” (string)
e = True (boolean)
```

Strings

```
x = “Python”
len(x)
x[0]
x[-1]
x[0:3]
```

Formatted strings

```
name = f”{first} {last}”
```

Escape sequences

```
\”
\’
\\
\n
```

String methods

```
x.upper()
x.lower()
x.title()
x.strip()
x.find(“p”)
x.replace(“a”, “b”)
“a” in x
```

Numer functions

```
round(x)
abs(x)
```

Type conversion

```
int(x)
float(x)
bool(x)
string(x)
```

Falsy values

```
0
“”
None
```

Conditional statements

```
if x == 1:
    print(“a”)
elif x == 2:
    print(“b”)
else:
    print(“c”)
```

Ternary operator

```
x = “a” if n > 1 else “b”
```

Boolean operators

```
x and y (both should be true)
x or y  (at least one true)
not x   (inverses a boolean)
```

Chaining comparison operators

```
if 18 <= age < 65:
```

For loops

```
for n in range(1, 10):
    …
```

While loops

```
while n > 10:
    …
```

Equality operators

```
== (equal)
!= (not equal)
```

Defining functions

```
def increment(number, by=1):
    return number + by
```

Keyword arguments

```
increment(2, by=1)
```

Variable number of arguments

```
def multiply(*numbers):
   for number in numbers:
      print number


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

Variable number of keyword arguments

```
def save_user(**user):
   …


save_user(id=1, name=“Mosh”)
```

Shortcuts

codewithmosh.com

DEBUGGING

Action	Shortcut
Start Debugging	F5
Step Over	F10
Step Into	F11
Step Out	Shift+F11
Stop Debugging	Shift+F5

CODING (Windows)

Action	Shortcut
End of line	End
Beginning of line	Home
End of file	Ctrl+End
Beginning of file	Ctrl+Home
Move line	Alt+Up/Down
Duplicate line	Shift+Alt+Down
Comment	Ctrl+/

CODING (Mac)

Action	Shortcut
End of line	fn+Right
Beginning of line	fn+Left
End of file	fn+Up
Beginning of file	fn+Down
Move line	Alt+Up/Down
Duplicate line	Shift+Alt+Down
Comment	Cmd+/

Lists

codewithmosh.com

Creating lists

```
letters = ["a", "b", "c"]
matrix = [[0, 1], [1, 2]]
zeros = [0] * 5
combined = zeros + letters
numbers = list(range(20))
```

Accessing items

```
letters = ["a", "b", "c", "d"]
letters[0]  # "a"
letters[-1] # "d"
```

Slicing lists

```
letters[0:3]   # "a", "b", "c"
letters[:3]    # "a", "b", "c"
letters[0:]    # "a", "b", "c", "d"
letters[:]     # "a", "b", "c", "d"
letters[::2]   # "a", "c"
letters[::-1]  # "d", "c", "b", "a"
```

Unpacking

```
first, second, *other = letters
```

Looping over lists

```
for letter in letters:
    ...

for index, letter in enumerate(letters):
    ...
```

Adding items

```
letters.append("e")
letters.insert(0, "-")
```

Removing items

```
letters.pop()
letters.pop(0)
letters.remove("b")
del letters[0:3]
```

Finding items

```
if "f" in letters:
    letters.index("f")
```

Sorting lists

```
letters.sort()
letters.sort(reverse=True)
```

Custom sorting

```
items = [
    ("Product1", 10),
    ("Product2", 9),
    ("Product3", 11)
]

items.sort(key=lambda item: item[1])
```

Zip function

```
list1 = [1, 2, 3]
list2 = [10, 20, 30]
combined = list(zip(list1, list2))
# [(1, 10), (2, 20)]
```

Unpacking operator

```
list1 = [1, 2, 3]
list2 = [10, 20, 30]
combined = [*list1, “a”, *list2]
```

Tuples, Sets, and Dictionaries

Tuples

```
point = 1, 2, 3
point = (1, 2, 3)
point = (1,)
point = ()
point(0:2)
x, y, z = point
if 10 in point:
    …
```

Swapping variables

```
x = 10
y = 11
x, y = y, x
```

Arrays

```
from array import array
numbers = array(“i”, [1, 2, 3])
```

Sets

```
first = {1, 2, 3, 4}
second = {1, 5}

first | second  # {1, 2, 3, 4, 5}
first & second  # {1}
first - second  # {2, 3, 4}
first ^ second  # {2, 3, 4, 5}
```

Dictionaries

```
point = {"x": 1, "y": 2}
point = dict(x=1, y=2)
point["z"] = 3
if "a" in point:
    ...
point.get("a", 0) # 0
del point["x"]
for key, value in point.items():
    ...
```

Comprehensions

List comprehensions

```
values = [x * 2 for x in range(5)]
values = [x * 2 for x in range(5) if x % 2 == 0]
```

Set comprehensions

```
values = {x * 2 for x in range(5)}
```

Dictionary comprehensions

```
values = {x: x * 2 for x in range(5)}
```

Generator expressions

```
values = {x: x * 2 for x in range(500000)}
```

Exceptions

Handling Exceptions

```
try:
  …
except (ValueError, ZeroDivisionError):
  …
else:
  # no exceptions raised
finally:
  # cleanup code
```

Raising exceptions

```
if x < 1:
    raise ValueError("…")
```

The with statement

```
with open("file.txt") as file:
   …
```

Creating classes

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def draw(self):
        …
```

Instance vs class attributes

```
class Point:
    default_color = “red”

    def __init__(self, x, y):
        self.x = x
```

Instance vs class methods

```
class Point:
    def draw(self):
        …

    @classmethod
    def zero(cls):
        return cls(0, 0)
```

Magic methods

```
__str__()
__eq__()
__cmp__()
```

Private members

```
class Point:
    def __init__(self, x):
        self.__x = x
```

Properties

```
class Point:
    def __init__(self, x):
        self.__x = x

    @property
    def x(self):
        return self.__x

    @property.setter:
    def x.setter(self, value):
        self.__x = value
```

Inheritance

```
class FileStream(Stream):
    def open(self):
        super().open()
        …
```

Multiple inheritance

```
class FlyingFish(Flyer, Swimmer):
    …
```

Abstract base classes

```
from abc import ABC, abstractmethod

class Stream(ABC):
    @abstractmethod
    def read(self):
        pass
```

Named tuples

```
from collections import namedtuple

Point = namedtuple(“Point”, [“x”, “y”])
point = Point(x=1, y=2)
```

Python For Data Science *Cheat Sheet*

Python Basics

Learn More Python for Data Science Interactively at www.datacamp.com

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
 5
```

Calculations With Variables

`>>> x+2` `7`	Sum of two variables
`>>> x-2` `3`	Subtraction of two variables
`>>> x*2` `10`	Multiplication of two variables
`>>> x**2` `25`	Exponentiation of a variable
`>>> x%2` `1`	Remainder of a variable
`>>> x/float(2)` `2.5`	Division of a variable

Types and Type Conversion

`str()`	`'5', '3.45', 'True'`	Variables to strings
`int()`	`5, 3, 1`	Variables to integers
`float()`	`5.0, 1.0`	Variables to floats
`bool()`	`True, True, True`	Variables to booleans

Asking For Help

```
>>> help(str)
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset

`>>> my_list[1]`	Select item at index 1
`>>> my_list[-3]`	Select 3rd last item

Slice

`>>> my_list[1:3]`	Select items at index 1 and 2
`>>> my_list[1:]`	Select items after index 0
`>>> my_list[:3]`	Select items before index 3
`>>> my_list[:]`	Copy my_list

Subset Lists of Lists

`>>> my_list2[1][0]` `>>> my_list2[1][:2]`	my_list[list][itemOfList]

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

`>>> my_list.index(a)`	Get the index of an item
`>>> my_list.count(a)`	Count an item
`>>> my_list.append('!')`	Append an item at a time
`>>> my_list.remove('!')`	Remove an item
`>>> del(my_list[0:1])`	Remove an item
`>>> my_list.reverse()`	Reverse the list
`>>> my_list.extend('!')`	Append an item
`>>> my_list.pop(-1)`	Remove an item
`>>> my_list.insert(0,'!')`	Insert an item
`>>> my_list.sort()`	Sort the list

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
 'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
 'thisStringIsAwesomeInnit'
>>> 'm' in my_string
 True
```

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

`>>> my_string.upper()`	String to uppercase
`>>> my_string.lower()`	String to lowercase
`>>> my_string.count('w')`	Count String elements
`>>> my_string.replace('e', 'i')`	Replace String elements
`>>> my_string.strip()`	Strip whitespaces

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

pandas — Data analysis
learn — Machine learning
NumPy — Scientific computing
matplotlib — 2D plotting

Install Python

Leading open data science platform powered by Python

Free IDE that is included with Anaconda

Create and share documents with live code, visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset

`>>> my_array[1]` `2`	Select item at index 1

Slice

`>>> my_array[0:2]` `array([1, 2])`	Select items at index 0 and 1

Subset 2D Numpy arrays

`>>> my_2darray[:,0]` `array([1, 4])`	my_2darray[rows, columns]

Numpy Array Operations

```
>>> my_array > 3
 array([False, False, False,  True], dtype=bool)
>>> my_array * 2
 array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
 array([6, 8, 10, 12])
```

Numpy Array Functions

`>>> my_array.shape`	Get the dimensions of the array
`>>> np.append(other_array)`	Append items to an array
`>>> np.insert(my_array, 1, 5)`	Insert items in an array
`>>> np.delete(my_array,[1])`	Delete items in an array
`>>> np.mean(my_array)`	Mean of the array
`>>> np.median(my_array)`	Median of the array
`>>> my_array.corrcoef()`	Correlation coefficient
`>>> np.std(my_array)`	Standard deviation

Python For Data Science *Cheat Sheet*

Jupyter Notebook

Learn More Python for Data Science Interactively at www.DataCamp.com

Saving/Loading Notebooks

File | Edit | View | Insert

- New Notebook — Create new notebook
- Open... — Open an existing notebook
- Make a Copy... — Make a copy of the current notebook
- Rename... — Rename notebook
- Save and Checkpoint — Save current notebook and record checkpoint
- Revert to Checkpoint — Revert notebook to a previous checkpoint
- Print Preview — Preview of the printed notebook
- Download as — Download notebook as
  - IPython notebook
  - Python
  - HTML
  - Markdown
  - reST
  - LaTeX
  - PDF
- Trusted Notebook
- Close and Halt — Close notebook & stop running any scripts

Writing Code And Text

Code and text are encapsulated by 3 basic cell types: markdown cells, code cells, and raw NBConvert cells.

Edit Cells

Edit | View | Insert | Cell

- Cut Cells — Cut currently selected cells to clipboard
- Copy Cells — Copy cells from clipboard to current cursor position
- Paste Cells Above — Paste cells from clipboard above current cell
- Paste Cells Below — Paste cells from clipboard below current cell
- Paste Cells & Replace — Paste cells from clipboard on top of current cel
- Delete Cells — Delete current cells
- Undo Delete Cells — Revert *"Delete Cells"* invocation
- Split Cell — Split up a cell from current cursor position
- Merge Cell Above — Merge current cell with the one above
- Merge Cell Below — Merge current cell with the one below
- Move Cell Up — Move current cell up
- Move Cell Down — Move current cell down
- Edit Notebook Metadata — Adjust metadata underlying the current notebook
- Find and Replace — Find and replace in selected cells
- Cut Cell Attachments — Remove cell attachments
- Copy Cell Attachments — Copy attachments of current cell
- Paste Cell Attachments — Paste attachments of current cell
- Insert Image — Insert image in selected cells

Insert Cells

Insert | Cell | Kernel

- Insert Cell Above — Add new cell above the current one
- Insert Cell Below — Add new cell below the current one

Working with Different Programming Languages

Kernels provide computation and communication with front-end interfaces like the notebooks. There are three main kernels:

IPython

IRkernel

IJulia

Installing Jupyter Notebook will automatically install the IPython kernel.

Kernel | Widgets | Help

- Interrupt — Interrupt kernel
- Restart — Restart kernel
- Restart & Clear Output — Interrupt kernel & clear all output
- Restart & Run All — Restart kernel & run all cells
- Reconnect — Connect back to a remote notebook
- Shutdown — Restart kernel & run all cells
- Change kernel — Run other installed kernels

Widgets

Notebook widgets provide the ability to visualize and control changes in your data, often as a control like a slider, textbox, etc.

You can use them to build interactive GUIs for your notebooks or to synchronize stateful and stateless information between Python and JavaScript.

Widgets | Help

- Save Notebook with Widgets — Save notebook with interactive widgets
- Download Widget State — Download serialized state of all widget models in use
- Embed Widgets — Embed current widgets

Command Mode:

jupyter MyJupyterNotebook Last Checkpoint: a few seconds ago (unsaved changes) — Logout (15)

File | Edit | View | Insert | Cell | Kernel | Widgets | Help — Trusted — Python 3 (13) ○ (14)

Toolbar: 1 2 3 4 5 6 7 8 9 10 11 (Code) 12

```
In [ ]:
```

Edit Mode:

```
In [ ]: |
```

1. Save and checkpoint
2. Insert cell below
3. Cut cell
4. Copy cell(s)
5. Paste cell(s) below
6. Move cell up
7. Move cell down
8. Run current cell
9. Interrupt kernel
10. Restart kernel
11. Display characteristics
12. Open command palette
13. Current kernel
14. Kernel status
15. Log out from notebook server

Executing Cells

Cell | Kernel | Widgets

- Run Cells — Run selected cell(s)
- Run Cells and Select Below — Run current cells down and create a new one below
- Run Cells and Insert Below — Run current cells down and create a new one above
- Run All — Run all cells
- Run All Above — Run all cells above the current cell
- Run All Below — Run all cells below the current cell
- Cell Type — Change the cell type of current cell
- Current Outputs — toggle, toggle scrolling and clear current outputs
- All Output — toggle, toggle scrolling and clear all output

View Cells

View | Insert | Cell

- Toggle Header — Toggle display of Jupyter logo and filename
- Toggle Toolbar — Toggle display of toolbar
- Toggle Line Numbers — Toggle line numbers in cells
- Cell Toolbar — Toggle display of cell action icons:
  - None
  - Edit metadata
  - Raw cell format
  - Slideshow
  - Attachments
  - Tags

Asking For Help

Help

- User Interface Tour — Walk through a UI tour
- Keyboard Shortcuts — List of built-in keyboard shortcuts
- Edit Keyboard Shortcuts — Edit the built-in keyboard shortcuts
- Notebook Help — Notebook help topics
- Markdown — Description of markdown available in notebook
- Jupyter-contrib nbextensions — Information on unofficial Jupyter Notebook extensions
- Python — Python help topics
- IPython — IPython help topics
- NumPy — NumPy help topics
- SciPy — SciPy help topics
- Matplotlib — Matplotlib help topics
- SymPy — SymPy help topics
- pandas — Pandas help topics
- About — About Jupyter Notebook

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

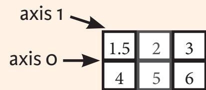
Use the following import convention:

```
>>> import numpy as np
```

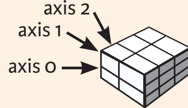
NumPy Arrays

1D array

2D array



3D array



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

Code	Description
`>>> np.zeros((3,4))`	Create an array of zeros
`>>> np.ones((2,3,4),dtype=np.int16)`	Create an array of ones
`>>> d = np.arange(10,25,5)`	Create an array of evenly spaced values (step value)
`>>> np.linspace(0,2,9)`	Create an array of evenly spaced values (number of samples)
`>>> e = np.full((2,2),7)`	Create a constant array
`>>> f = np.eye(2)`	Create a 2X2 identity matrix
`>>> np.random.random((2,2))`	Create an array with random values
`>>> np.empty((3,2))`	Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

Code	Description
`>>> np.int64`	Signed 64-bit integer types
`>>> np.float32`	Standard double-precision floating point
`>>> np.complex`	Complex numbers represented by 128 floats
`>>> np.bool`	Boolean type storing `TRUE` and `FALSE` values
`>>> np.object`	Python object type
`>>> np.string_`	Fixed-length string type
`>>> np.unicode_`	Fixed-length unicode type

Inspecting Your Array

Code	Description
`>>> a.shape`	Array dimensions
`>>> len(a)`	Length of array
`>>> b.ndim`	Number of array dimensions
`>>> e.size`	Number of array elements
`>>> b.dtype`	Data type of array elements
`>>> b.dtype.name`	Name of data type
`>>> b.astype(int)`	Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

Code	Description

`>>> g = a - b`
`array([[-0.5, 0. , 0. ],`
`[-3. , -3. , -3. ]])`
	Subtraction

`>>> np.subtract(a,b)`	Subtraction

`>>> b + a`
`array([[ 2.5, 4. , 6. ],`
`[ 5. , 7. , 9. ]])`
	Addition

`>>> np.add(b,a)`	Addition

`>>> a / b`
`array([[ 0.66666667, 1. , 1. ],`
`[ 0.25 , 0.4 , 0.5 ]])`
	Division

`>>> np.divide(a,b)`	Division

`>>> a * b`
`array([[ 1.5, 4. , 9. ],`
`[ 4. , 10. , 18. ]])`
	Multiplication

`>>> np.multiply(a,b)`	Multiplication
`>>> np.exp(b)`	Exponentiation
`>>> np.sqrt(b)`	Square root
`>>> np.sin(a)`	Print sines of an array
`>>> np.cos(b)`	Element-wise cosine
`>>> np.log(a)`	Element-wise natural logarithm

`>>> e.dot(f)`
`array([[ 7., 7.],`
`[ 7., 7.]])`
	Dot product

Comparison

Code	Description

`>>> a == b`
`array([[False, True, True],`
`[False, False, False]], dtype=bool)`
	Element-wise comparison

`>>> a < 2` `array([True, False, False], dtype=bool)`	Element-wise comparison
`>>> np.array_equal(a, b)`	Array-wise comparison

Aggregate Functions

Code	Description
`>>> a.sum()`	Array-wise sum
`>>> a.min()`	Array-wise minimum value
`>>> b.max(axis=0)`	Maximum value of an array row
`>>> b.cumsum(axis=1)`	Cumulative sum of the elements
`>>> a.mean()`	Mean
`>>> b.median()`	Median
`>>> a.corrcoef()`	Correlation coefficient
`>>> np.std(b)`	Standard deviation

Copying Arrays

Code	Description
`>>> h = a.view()`	Create a view of the array with the same data
`>>> np.copy(a)`	Create a copy of the array
`>>> h = a.copy()`	Create a deep copy of the array

Sorting Arrays

Code	Description
`>>> a.sort()`	Sort an array
`>>> c.sort(axis=0)`	Sort the elements of an array's axis

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

Code	Description
`>>> a[2]` `3`	Select the element at the 2nd index
`>>> b[1,2]` `6.0`	Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing

Code	Description
`>>> a[0:2]` `array([1, 2])`	Select items at index 0 and 1
`>>> b[0:2,1]` `array([ 2., 5.])`	Select items at rows 0 and 1 in column 1
`>>> b[:1]` `array([[1.5, 2., 3.]])`	Select all items at row 0 (equivalent to `b[0:1, :]`)

`>>> c[1,...]`
`array([[[ 3., 2., 1.],`
`[ 4., 5., 6.]]])`
	Same as `[1,:,:]`

`>>> a[ : :-1]` `array([3, 2, 1])`	Reversed array `a`

Boolean Indexing

Code	Description
`>>> a[a<2]` `array([1])`	Select elements from `a` less than 2

Fancy Indexing

Code	Description
`>>> b[[1, 0, 1, 0], [0, 1, 2, 0]]` `array([ 4. , 2. , 6. , 1.5])`	Select elements (1,0),(0,1),(1,2) and (0,0)

`>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]`
`array([[ 4. ,5. , 6. , 4. ],`
`[ 1.5, 2. , 3. , 1.5],`
`[ 4. , 5. , 6. , 4. ],`
`[ 1.5, 2. , 3. , 1.5]])`
	Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

Code	Description
`>>> i = np.transpose(b)`	Permute array dimensions
`>>> i.T`	Permute array dimensions

Changing Array Shape

Code	Description
`>>> b.ravel()`	Flatten the array
`>>> g.reshape(3,-2)`	Reshape, but don't change data

Adding/Removing Elements

Code	Description
`>>> h.resize((2,6))`	Return a new array with shape (2,6)
`>>> np.append(h,g)`	Append items to an array
`>>> np.insert(a, 1, 5)`	Insert items in an array
`>>> np.delete(a,[1])`	Delete items from an array

Combining Arrays

Code	Description
`>>> np.concatenate((a,d),axis=0)` `array([ 1, 2, 3, 10, 15, 20])`	Concatenate arrays

`>>> np.vstack((a,b))`
`array([[ 1. , 2. , 3. ],`
`[ 1.5, 2. , 3. ],`
`[ 4. , 5. , 6. ]])`
	Stack arrays vertically (row-wise)

`>>> np.r_[e,f]`	Stack arrays vertically (row-wise)

`>>> np.hstack((e,f))`
`array([[ 7., 7., 1., 0.],`
`[ 7., 7., 0., 1.]])`
	Stack arrays horizontally (column-wise)

`>>> np.column_stack((a,d))`
`array([[ 1, 10],`
`[ 2, 15],`
`[ 3, 20]])`
	Create stacked column-wise arrays

`>>> np.c_[a,d]`	Create stacked column-wise arrays

Splitting Arrays

Code	Description
`>>> np.hsplit(a,3)` `[array([1]),array([2]),array([3])]`	Split the array horizontally at the 3rd index

`>>> np.vsplit(c,2)`
`[array([[[ 1.5, 2. , 1. ],`
`[ 4. , 5. , 6. ]]]),`
`array([[[ 3., 2., 3.],`
`[ 4., 5., 6.]]])]`
	Split the array vertically at the 2nd index

DataCamp
Learn Python for Data Science **Interactively**

Python For Data Science *Cheat Sheet*

SciPy - Linear Algebra

Learn More Python for Data Science Interactively at www.datacamp.com

SciPy

The **SciPy** library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.

Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j,2j,3j), (4j,5j,6j)])
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]])
```

Index Tricks

Command	Description
`>>> np.mgrid[0:5,0:5]`	Create a dense meshgrid
`>>> np.ogrid[0:2,0:2]`	Create an open meshgrid
`>>> np.r_[[3,[0]*5,-1:1:10j]`	Stack arrays vertically (row-wise)
`>>> np.c_[b,c]`	Create stacked column-wise arrays

Shape Manipulation

Command	Description
`>>> np.transpose(b)`	Permute array dimensions
`>>> b.flatten()`	Flatten the array
`>>> np.hstack((b,c))`	Stack arrays horizontally (column-wise)
`>>> np.vstack((a,b))`	Stack arrays vertically (row-wise)
`>>> np.hsplit(c,2)`	Split the array horizontally at the 2nd index
`>>> np.vpslit(d,2)`	Split the array vertically at the 2nd index

Polynomials

```
>>> from numpy import poly1d
>>> p = poly1d([3,4,5])
```

Create a polynomial object

Vectorizing Functions

```
>>> def myfunc(a):
        if a < 0:
          return a*2
        else:
          return a/2
>>> np.vectorize(myfunc)
```

Vectorize functions

Type Handling

Command	Description
`>>> np.real(c)`	Return the real part of the array elements
`>>> np.imag(c)`	Return the imaginary part of the array elements
`>>> np.real_if_close(c,tol=1000)`	Return a real array if complex parts close to 0
`>>> np.cast['f'](np.pi)`	Cast object to a data type

Other Useful Functions

Command	Description
`>>> np.angle(b,deg=True)`	Return the angle of the complex argument
`>>> g = np.linspace(0,np.pi,num=5)`	Create an array of evenly spaced values (number of samples)
`>>> g [3:] += np.pi`	
`>>> np.unwrap(g)`	Unwrap
`>>> np.logspace(0,10,3)`	Create an array of evenly spaced values (log scale)
`>>> np.select([c<4],[c*2])`	Return values from a list of arrays depending on conditions
`>>> misc.factorial(a)`	Factorial
`>>> misc.comb(10,3,exact=True)`	Combine N things taken at k time
`>>> misc.central_diff_weights(3)`	Weights for Np-point central derivative
`>>> misc.derivative(myfunc,1.0)`	Find the n-th derivative of a function at a point

Linear Algebra

Also see NumPy

You'll use the `linalg` and `sparse` modules. Note that `scipy.linalg` contains and expands on `numpy.linalg`.

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Command	Description
Inverse	
`>>> A.I`	Inverse
`>>> linalg.inv(A)`	Inverse
`>>> A.T`	Tranpose matrix
`>>> A.H`	Conjugate transposition
`>>> np.trace(A)`	Trace
Norm	
`>>> linalg.norm(A)`	Frobenius norm
`>>> linalg.norm(A,1)`	L1 norm (max column sum)
`>>> linalg.norm(A,np.inf)`	L inf norm (max row sum)
Rank	
`>>> np.linalg.matrix_rank(C)`	Matrix rank
Determinant	
`>>> linalg.det(A)`	Determinant
Solving linear problems	
`>>> linalg.solve(A,b)`	Solver for dense matrices
`>>> E = np.mat(a).T`	Solver for dense matrices
`>>> linalg.lstsq(D,E)`	Least-squares solution to linear matrix equation
Generalized inverse	
`>>> linalg.pinv(C)`	Compute the pseudo-inverse of a matrix (least-squares solver)
`>>> linalg.pinv2(C)`	Compute the pseudo-inverse of a matrix (SVD)

Creating Sparse Matrices

Command	Description
`>>> F = np.eye(3, k=1)`	Create a 2X2 identity matrix
`>>> G = np.mat(np.identity(2))`	Create a 2x2 identity matrix
`>>> C[C > 0.5] = 0`	
`>>> H = sparse.csr_matrix(C)`	Compressed Sparse Row matrix
`>>> I = sparse.csc_matrix(D)`	Compressed Sparse Column matrix
`>>> J = sparse.dok_matrix(A)`	Dictionary Of Keys matrix
`>>> E.todense()`	Sparse matrix to full matrix
`>>> sparse.isspmatrix_csc(A)`	Identify sparse matrix

Sparse Matrix Routines

Command	Description
Inverse	
`>>> sparse.linalg.inv(I)`	Inverse
Norm	
`>>> sparse.linalg.norm(I)`	Norm
Solving linear problems	
`>>> sparse.linalg.spsolve(H,I)`	Solver for sparse matrices

Sparse Matrix Functions

Command	Description
`>>> sparse.linalg.expm(I)`	Sparse matrix exponential

Matrix Functions

Command	Description
Addition	
`>>> np.add(A,D)`	Addition
Subtraction	
`>>> np.subtract(A,D)`	Subtraction
Division	
`>>> np.divide(A,D)`	Division
Multiplication	
`>>> np.multiply(D,A)`	Multiplication
`>>> np.dot(A,D)`	Dot product
`>>> np.vdot(A,D)`	Vector dot product
`>>> np.inner(A,D)`	Inner product
`>>> np.outer(A,D)`	Outer product
`>>> np.tensordot(A,D)`	Tensor dot product
`>>> np.kron(A,D)`	Kronecker product
Exponential Functions	
`>>> linalg.expm(A)`	Matrix exponential
`>>> linalg.expm2(A)`	Matrix exponential (Taylor Series)
`>>> linalg.expm3(D)`	Matrix exponential (eigenvalue decomposition)
Logarithm Function	
`>>> linalg.logm(A)`	Matrix logarithm
Trigonometric Tunctions	
`>>> linalg.sinm(D)`	Matrix sine
`>>> linalg.cosm(D)`	Matrix cosine
`>>> linalg.tanm(A)`	Matrix tangent
Hyperbolic Trigonometric Functions	
`>>> linalg.sinhm(D)`	Hypberbolic matrix sine
`>>> linalg.coshm(D)`	Hyperbolic matrix cosine
`>>> linalg.tanhm(A)`	Hyperbolic matrix tangent
Matrix Sign Function	
`>>> np.sigm(A)`	Matrix sign function
Matrix Square Root	
`>>> linalg.sqrtm(A)`	Matrix square root
Arbitrary Functions	
`>>> linalg.funm(A, lambda x: x*x)`	Evaluate matrix function

Decompositions

Command	Description
Eigenvalues and Eigenvectors	
`>>> la, v = linalg.eig(A)`	Solve ordinary or generalized eigenvalue problem for square matrix
`>>> l1, l2 = la`	Unpack eigenvalues
`>>> v[:,0]`	First eigenvector
`>>> v[:,1]`	Second eigenvector
`>>> linalg.eigvals(A)`	Unpack eigenvalues
Singular Value Decomposition	
`>>> U,s,Vh = linalg.svd(B)`	Singular Value Decomposition (SVD)
`>>> M,N = B.shape`	
`>>> Sig = linalg.diagsvd(s,M,N)`	Construct sigma matrix in SVD
LU Decomposition	
`>>> P,L,U = linalg.lu(C)`	LU Decomposition

Sparse Matrix Decompositions

Command	Description
`>>> la, v = sparse.linalg.eigs(F,1)`	Eigenvalues and eigenvectors
`>>> sparse.linalg.svds(H, 2)`	SVD

Asking For Help

```
>>> help(scipy.linalg.diagsvd)
>>> np.info(np.matrix)
```

DataCamp

Learn Python for Data Science Interactively

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

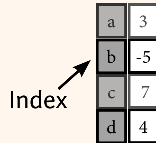
Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

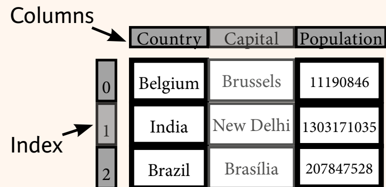
Series

A **one-dimensional** labeled array capable of holding any data type




```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



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

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> pd.to_sql('myDf', engine)
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

Code	Description
`>>> s['b']` → `-5`	Get one element
`>>> df[1:]`	Get subset of a DataFrame

```
>>> s['b']
 -5

>>> df[1:]
   Country     Capital  Population
 1   India   New Delhi  1303171035
 2  Brazil    Brasília   207847528
```

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc([0],[0])
 'Belgium'
>>> df.iat([0],[0])
 'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc([0], ['Country'])
 'Belgium'
>>> df.at([0], ['Country'])
 'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
 Country       Brazil
 Capital     Brasília
 Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:,'Capital']
 0     Brussels
 1    New Delhi
 2     Brasília
```

Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
 'New Delhi'
```

Select rows and columns

Boolean Indexing

Code	Description
`>>> s[~(s > 1)]`	Series `s` where value is not >1
`>>> s[(s < -1) | (s > 2)]`	`s` where value is <-1 or >2
`>>> df[df['Population']>1200000000]`	Use filter to adjust DataFrame

Setting

Code	Description
`>>> s['a'] = 6`	Set index `a` of Series `s` to 6

Dropping

Code	Description
`>>> s.drop(['a', 'c'])`	Drop values from rows (axis=0)
`>>> df.drop('Country', axis=1)`	Drop values from columns(axis=1)

Sort & Rank

Code	Description
`>>> df.sort_index()`	Sort by labels along an axis
`>>> df.sort_values(by='Country')`	Sort by the values along an axis
`>>> df.rank()`	Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

Code	Description
`>>> df.shape`	(rows,columns)
`>>> df.index`	Describe index
`>>> df.columns`	Describe DataFrame columns
`>>> df.info()`	Info on DataFrame
`>>> df.count()`	Number of non-NA values

Summary

Code	Description
`>>> df.sum()`	Sum of values
`>>> df.cumsum()`	Cummulative sum of values
`>>> df.min()/df.max()`	Minimum/maximum values
`>>> df.idxmin()/df.idxmax()`	Minimum/Maximum index value
`>>> df.describe()`	Summary statistics
`>>> df.mean()`	Mean of values
`>>> df.median()`	Median of values

Applying Functions

Code	Description
`>>> f = lambda x: x*2`	
`>>> df.apply(f)`	Apply function
`>>> df.applymap(f)`	Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
 a    10.0
 b     NaN
 c     5.0
 d     7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
 a    10.0
 b    -5.0
 c     5.0
 d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

Python For Data Science *Cheat Sheet*

Scikit-Learn

Learn Python for data science **Interactively** at www.DataCamp.com

Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.

A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M','M','F','F','M','F','M','M','F','F','F'])
>>> X[X < 0.7] = 0
```

Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
                                                        y,
                                                        random_state=0)
```

Preprocessing The Data

Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

Encoding Categorical Features

```
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> y = enc.fit_transform(y)
```

Imputing Missing Values

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit_transform(X_train)
```

Generating Polynomial Features

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(5)
>>> poly.fit_transform(X)
```

Create Your Model

Supervised Learning Estimators

Linear Regression

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

Support Vector Machines (SVM)

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

Naive Bayes

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

KNN

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

Unsupervised Learning Estimators

Principal Component Analysis (PCA)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

K Means

```
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

Model Fitting

Supervised learning

Code	Description
`>>> lr.fit(X, y)`	Fit the model to the data
`>>> knn.fit(X_train, y_train)`	
`>>> svc.fit(X_train, y_train)`	

Unsupervised Learning

Code	Description
`>>> k_means.fit(X_train)`	Fit the model to the data
`>>> pca_model = pca.fit_transform(X_train)`	Fit to data, then transform it

Prediction

Supervised Estimators

Code	Description
`>>> y_pred = svc.predict(np.random.random((2,5)))`	Predict labels
`>>> y_pred = lr.predict(X_test)`	Predict labels
`>>> y_pred = knn.predict_proba(X_test)`	Estimate probability of a label

Unsupervised Estimators

Code	Description
`>>> y_pred = k_means.predict(X_test)`	Predict labels in clustering algos

Evaluate Your Model's Performance

Classification Metrics

Accuracy Score

```
>>> knn.score(X_test, y_test)
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
```

Estimator score method

Metric scoring functions

Classification Report

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, y_pred))
```

Precision, recall, f1-score and support

Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

Regression Metrics

Mean Absolute Error

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

Mean Squared Error

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

R^2 Score

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

Clustering Metrics

Adjusted Rand Index

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

Homogeneity

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

V-measure

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

Tune Your Model

Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
              "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
                        param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5),
              "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn,
                                 param_distributions=params,
                                 cv=4,
                                 n_iter=8,
                                 random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```

Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python Interactively at www.DataCamp.com

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

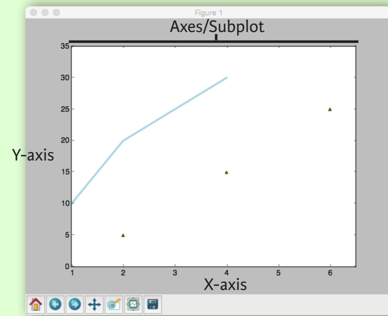
Axes

All plotting is done with respect to an `Axes`. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data 2 Create plot 3 Plot 4 Customize plot 5 Save plot 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]                                    Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure()                               Step 2
>>> ax = fig.add_subplot(111)                        Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3)    Step 3, 4
>>> ax.scatter([2,4,6],
               [5,15,25],
               color='darkgreen',
               marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()                                       Step 6
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                   cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(8, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)                        Add padding to a plot
>>> ax.axis('equal')                               Set the aspect ratio of the plot to 1
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])          Set limits for x-and y-axis
>>> ax.set_xlim(0,10.5)                            Set limits for x-axis
```

Legends

```
>>> ax.set(title='An Example Axes',                Set a title and x-and y-axis labels
           ylabel='Y-Axis',
           xlabel='X-Axis')
>>> ax.legend(loc='best')                          No overlapping plot elements
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),                 Manually set x-ticks
                 ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',                       Make y-ticks longer and go in and out
                   direction='inout',
                   length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,               Adjust the spacing between subplots
                         hspace=0.3,
                         left=0.125,
                         right=0.9,
                         top=0.9,
                         bottom=0.1)
>>> fig.tight_layout()                             Fit subplot(s) in to the figure area
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)           Make the top axis line for a plot invisible
>>> ax1.spines['bottom'].set_position(('outward',10))  Move the bottom axis line outward
```

3 Plotting Routines

1D Data

```
>>> fig, ax = plt.subplots()
>>> lines = ax.plot(x,y)                           Draw points with lines or markers connecting them
>>> ax.scatter(x,y)                                Draw unconnected points, scaled or colored
>>> axes[0,0].bar([1,2,3],[3,4,5])                 Plot vertical rectangles (constant width)
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])            Plot horiontal rectangles (constant height)
>>> axes[1,1].axhline(0.45)                        Draw a horizontal line across axes
>>> axes[0,1].axvline(0.65)                        Draw a vertical line across axes
>>> ax.fill(x,y,color='blue')                      Draw filled polygons
>>> ax.fill_between(x,y,color='yellow')            Fill between y-values and 0
```

Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)                   Add an arrow to the axes
>>> axes[1,1].quiver(y,z)                          Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V)                  Plot a 2D field of arrows
```

Data Distributions

```
>>> ax1.hist(y)                                    Plot a histogram
>>> ax3.boxplot(y)                                 Make a box and whisker plot
>>> ax3.violinplot(z)                              Make a violin plot
```

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,                            Colormapped or RGB arrays
                   cmap='gist_earth',
                   interpolation='nearest',
                   vmin=-2,
                   vmax=2)
```

```
>>> axes2[0].pcolor(data2)                         Pseudocolor plot of 2D array
>>> axes2[0].pcolormesh(data)                      Pseudocolor plot of 2D array
>>> CS = plt.contour(Y,X,U)                        Plot contours
>>> axes2[2].contourf(data1)                       Plot filled contours
>>> axes2[2]= ax.clabel(CS)                        Label a contour plot
```

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

```
>>> plt.cla()                                      Clear an axis
>>> plt.clf()                                      Clear the entire figure
>>> plt.close()                                    Close a window
```

DataCamp

Learn Python for Data Science Interactively

Python For Data Science *Cheat Sheet*

Seaborn

Learn Data Science **Interactively** at www.DataCamp.com

Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on matplotlib and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

1. Prepare some data
2. Control figure aesthetics
3. Plot with Seaborn
4. Further customize your plot

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")          Step 1
>>> sns.set_style("whitegrid")               Step 2
>>> g = sns.lmplot(x="tip",                  Step 3
                   y="total_bill",
                   data=tips,
                   aspect=2)
>>> g = (g.set_axis_labels("Tip","Total bill(USD)").
set(xlim=(0,10),ylim=(0,100)))               Step 4
>>> plt.title("title")
>>> plt.show(g)                              Step 5
```

1 Data

Also see Lists, NumPy & Pandas

```
>>> import pandas as pd
>>> import numpy as np
>>> uniform_data = np.random.rand(10, 12)
>>> data = pd.DataFrame({'x':np.arange(1,101),
                         'y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")
>>> iris = sns.load_dataset("iris")
```

2 Figure Aesthetics

Also see Matplotlib

Code	Description
`>>> f, ax = plt.subplots(figsize=(5,6))`	Create a figure and one subplot

Seaborn styles

Code	Description
`>>> sns.set()`	(Re)set the seaborn default
`>>> sns.set_style("whitegrid")`	Set the matplotlib parameters
`>>> sns.set_style("ticks", {"xtick.major.size":8, "ytick.major.size":8})`	Set the matplotlib parameters
`>>> sns.axes_style("whitegrid")`	Return a dict of params or use with `with` to temporarily set the style

Context Functions

Code	Description
`>>> sns.set_context("talk")`	Set context to `"talk"`
`>>> sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth":2.5})`	Set context to `"notebook"`, scale font elements and override param mapping

Color Palette

Code	Description
`>>> sns.set_palette("husl",3)`	Define the color palette
`>>> sns.color_palette("husl")`	Use with `with` to temporarily set palette
`>>> flatui = ["#9b59b6","#3498db","#95a5a6","#e74c3c","#34495e","#2ecc71"]`	
`>>> sns.set_palette(flatui)`	Set your own color palette

3 Plotting With Seaborn

Axis Grids

Code	Description
`>>> g = sns.FacetGrid(titanic, col="survived", row="sex")`	Subplot grid for plotting conditional relationships
`>>> g = g.map(plt.hist,"age")`	
`>>> sns.factorplot(x="pclass", y="survived", hue="sex", data=titanic)`	Draw a categorical plot onto a Facetgrid
`>>> sns.lmplot(x="sepal_width", y="sepal_length", hue="species", data=iris)`	Plot data and regression model fits across a FacetGrid
`>>> h = sns.PairGrid(iris)`	Subplot grid for plotting pairwise relationships
`>>> h = h.map(plt.scatter)`	
`>>> sns.pairplot(iris)`	Plot pairwise bivariate distributions
`>>> i = sns.JointGrid(x="x", y="y", data=data)`	Grid for bivariate plot with marginal univariate plots
`>>> i = i.plot(sns.regplot, sns.distplot)`	
`>>> sns.jointplot("sepal_length", "sepal_width", data=iris, kind='kde')`	Plot bivariate distribution

Categorical Plots

Scatterplot

Code	Description
`>>> sns.stripplot(x="species", y="petal_length", data=iris)`	Scatterplot with one categorical variable
`>>> sns.swarmplot(x="species", y="petal_length", data=iris)`	Categorical scatterplot with non-overlapping points

Bar Chart

Code	Description
`>>> sns.barplot(x="sex", y="survived", hue="class", data=titanic)`	Show point estimates and confidence intervals with scatterplot glyphs

Count Plot

Code	Description
`>>> sns.countplot(x="deck", data=titanic, palette="Greens_d")`	Show count of observations

Point Plot

Code	Description
`>>> sns.pointplot(x="class", y="survived", hue="sex", data=titanic, palette={"male":"g", "female":"m"}, markers=["^","o"], linestyles=["-","--"])`	Show point estimates and confidence intervals as rectangular bars

Boxplot

Code	Description
`>>> sns.boxplot(x="alive", y="age", hue="adult_male", data=titanic)`	Boxplot
`>>> sns.boxplot(data=iris,orient="h")`	Boxplot with wide-form data

Violinplot

Code	Description
`>>> sns.violinplot(x="age", y="sex", hue="survived", data=titanic)`	Violin plot

Regression Plots

Code	Description
`>>> sns.regplot(x="sepal_width", y="sepal_length", data=iris, ax=ax)`	Plot data and a linear regression model fit

Distribution Plots

Code	Description
`>>> plot = sns.distplot(data.y, kde=False, color="b")`	Plot univariate distribution

Matrix Plots

Code	Description
`>>> sns.heatmap(uniform_data,vmin=0,vmax=1)`	Heatmap

4 Further Customizations

Also see Matplotlib

Axisgrid Objects

Code	Description
`>>> g.despine(left=True)`	Remove left spine
`>>> g.set_ylabels("Survived")`	Set the labels of the y-axis
`>>> g.set_xticklabels(rotation=45)`	Set the tick labels for x
`>>> g.set_axis_labels("Survived", "Sex")`	Set the axis labels
`>>> h.set(xlim=(0,5), ylim=(0,5), xticks=[0,2.5,5], yticks=[0,2.5,5])`	Set the limit and ticks of the x-and y-axis

Plot

Code	Description
`>>> plt.title("A Title")`	Add plot title
`>>> plt.ylabel("Survived")`	Adjust the label of the y-axis
`>>> plt.xlabel("Sex")`	Adjust the label of the x-axis
`>>> plt.ylim(0,100)`	Adjust the limits of the y-axis
`>>> plt.xlim(0,10)`	Adjust the limits of the x-axis
`>>> plt.setp(ax,yticks=[0,5])`	Adjust a plot property
`>>> plt.tight_layout()`	Adjust subplot params

5 Show or Save Plot

Also see Matplotlib

Code	Description
`>>> plt.show()`	Show the plot
`>>> plt.savefig("foo.png")`	Save the plot as a figure
`>>> plt.savefig("foo.png", transparent=True)`	Save transparent figure

Close & Clear

Also see Matplotlib

Code	Description
`>>> plt.cla()`	Clear an axis
`>>> plt.clf()`	Clear an entire figure
`>>> plt.close()`	Close a window

DataCamp
Learn Python for Data Science **Interactively**

Python For Data Science *Cheat Sheet*

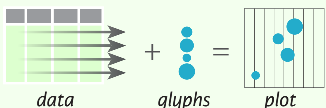
Bokeh

Learn Bokeh **Interactively** at www.DataCamp.com, taught by Bryan Van de Ven, core contributor

Plotting With Bokeh

The Python interactive visualization library **Bokeh** enables high-performance visual presentation of large datasets in modern web browsers.

Bokeh's mid-level general purpose `bokeh.plotting` interface is centered around two main components: data and glyphs.



The basic steps to creating plots with the `bokeh.plotting` interface are:

1. Prepare some data:
Python lists, NumPy arrays, Pandas DataFrames and other sequences of values
2. Create a new plot
3. Add renderers for your data, with visual customizations
4. Specify where to generate the output
5. Show or save the results

```
>>> from bokeh.plotting import figure
>>> from bokeh.io import output_file, show
>>> x = [1, 2, 3, 4, 5]                    Step 1
>>> y = [6, 7, 2, 4, 5]
>>> p = figure(title="simple line example",    Step 2
               x_axis_label='x',
               y_axis_label='y')
>>> p.line(x, y, legend="Temp.", line_width=2)    Step 3
>>> output_file("lines.html")    Step 4
>>> show(p)    Step 5
```

1 Data

Also see Lists, NumPy & Pandas

Under the hood, your data is converted to Column Data Sources. You can also do this manually:

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(np.array([[33.9,4,65, 'US'],
                                [32.4,4,66, 'Asia'],
                                [21.4,4,109, 'Europe']]),
                      columns=['mpg','cyl', 'hp', 'origin'],
                      index=['Toyota', 'Fiat', 'Volvo'])
```

```
>>> from bokeh.models import ColumnDataSource
>>> cds_df = ColumnDataSource(df)
```

2 Plotting

```
>>> from bokeh.plotting import figure
>>> p1 = figure(plot_width=300, tools='pan,box_zoom')
>>> p2 = figure(plot_width=300, plot_height=300,
                x_range=(0, 8), y_range=(0, 8))
>>> p3 = figure()
```

3 Renderers & Visual Customizations

Glyphs

Scatter Markers

```
>>> p1.circle(np.array([1,2,3]), np.array([3,2,1]),
              fill_color='white')
>>> p2.square(np.array([1.5,3.5,5.5]), [1,4,3],
              color='blue', size=1)
```

Line Glyphs

```
>>> p1.line([1,2,3,4], [3,4,5,6], line_width=2)
>>> p2.multi_line(pd.DataFrame([[1,2,3],[5,6,7]]),
                  pd.DataFrame([[3,4,5],[3,2,1]]),
                  color="blue")
```

Customized Glyphs

Also see Data

Selection and Non-Selection Glyphs

```
>>> p = figure(tools='box_select')
>>> p.circle('mpg', 'cyl', source=cds_df,
             selection_color='red',
             nonselection_alpha=0.1)
```

Hover Glyphs

```
>>> from bokeh.models import HoverTool
>>> hover = HoverTool(tooltips=None, mode='vline')
>>> p3.add_tools(hover)
```

Colormapping

```
>>> from bokeh.models import CategoricalColorMapper
>>> color_mapper = CategoricalColorMapper(
                      factors=['US', 'Asia', 'Europe'],
                      palette=['blue', 'red', 'green'])
>>> p3.circle('mpg', 'cyl', source=cds_df,
              color=dict(field='origin',
                         transform=color_mapper),
              legend='Origin')
```

Legend Location

Inside Plot Area

```
>>> p.legend.location = 'bottom_left'
```

Outside Plot Area

```
>>> from bokeh.models import Legend
>>> r1 = p2.asterisk(np.array([1,2,3]), np.array([3,2,1])
>>> r2 = p2.line([1,2,3,4], [3,4,5,6])
>>> legend = Legend(items=[("One" ,[p1, r1]),("Two",[r2])],
                    location=(0, -30))
>>> p.add_layout(legend, 'right')
```

Legend Orientation

```
>>> p.legend.orientation = "horizontal"
>>> p.legend.orientation = "vertical"
```

Legend Background & Border

```
>>> p.legend.border_line_color = "navy"
>>> p.legend.background_fill_color = "white"
```

Rows & Columns Layout

Rows

```
>>> from bokeh.layouts import row
>>> layout = row(p1,p2,p3)
```

Columns

```
>>> from bokeh.layouts import columns
>>> layout = column(p1,p2,p3)
```

Nesting Rows & Columns

```
>>>layout = row(column(p1,p2), p3)
```

Grid Layout

```
>>> from bokeh.layouts import gridplot
>>> row1 = [p1,p2]
>>> row2 = [p3]
>>> layout = gridplot([[p1,p2],[p3]])
```

Tabbed Layout

```
>>> from bokeh.models.widgets import Panel, Tabs
>>> tab1 = Panel(child=p1, title="tab1")
>>> tab2 = Panel(child=p2, title="tab2")
>>> layout = Tabs(tabs=[tab1, tab2])
```

Linked Plots

Linked Axes

```
>>> p2.x_range = p1.x_range
>>> p2.y_range = p1.y_range
```

Linked Brushing

```
>>> p4 = figure(plot_width = 100,
                tools='box_select,lasso_select')
>>> p4.circle('mpg', 'cyl', source=cds_df)
>>> p5 = figure(plot_width = 200,
                tools='box_select,lasso_select')
>>> p5.circle('mpg', 'hp', source=cds_df)
>>> layout = row(p4,p5)
```

4 Output & Export

Notebook

```
>>> from bokeh.io import output_notebook, show
>>> output_notebook()
```

HTML

Standalone HTML

```
>>> from bokeh.embed import file_html
>>> from bokeh.resources import CDN
>>> html = file_html(p, CDN, "my_plot")
```

```
>>> from bokeh.io import output_file, show
>>> output_file('my_bar_chart.html', mode='cdn')
```

Components

```
>>> from bokeh.embed import components
>>> script, div = components(p)
```

PNG

```
>>> from bokeh.io import export_png
>>> export_png(p, filename="plot.png")
```

SVG

```
>>> from bokeh.io import export_svgs
>>> p.output_backend = "svg"
>>> export_svgs(p, filename="plot.svg")
```

5 Show or Save Your Plots

```
>>> show(p1)
>>> save(p1)
```

```
>>> show(layout)
>>> save(layout)
```