Python Bookdown

Yong Keh Soon

2020-12-27

Contents

Introduction

Following text added by Alfonso R. Reyes

Main sections of the book

- Fundamentals
- numpy
- pandas
- Visualization
- sklearn
- Natural Language Processing
- Web scrapping
- Finance

Python environment

You may need to create a Python environment that covers all packages and dependencies for building this book. Once you have Anaconda3 installed in your computer, creating the Python environment is very easy. Go to your tterminal and run:

```
conda env create -f environment.yml
```

Anaconda will read the file listing the core dependencies, and install them following the package version specified. This is what is inside `environment.yml`:

```
name: python_book
channels:
  - anaconda
  - conda-forge
  - defaults
dependencies:
  - python=3.7
  - beautifulsoup4=4.9.3
  - matplotlib=3.3.1
```

```
  - nltk=3.5
  - numpy=1.19.1
  - pandas=1.1.3
  - pandas-datareader=0.9.0
  - pip=20.2.4
  - requests=2.24.0
  - scikit-learn=0.23.2
  - seaborn=0.11.0
  - urllib3=1.25.11
  - pip:
    - cufflinks
    - h5py==2.10.0
    - nlpia==0.5.2
    - plotnine==0.7
    - plydata==0.4.2
    - yfinance==0.1.55
prefix: /home/msfz751/anaconda3/envs/python_book
```

Automating the builds with a `Makefile`

Rules for building the book

Here are couple of rules from the `Makefile`:

```
# knit the book and then open it in the browser
.PHONY: gitbook1 gitbook2
gitbook1: build_book1 open_book

gitbook2: build_book2 open_book

# use rstudio pandoc
# this rule sets the PANDOC environment variable from the shell
build_book1:
    export RSTUDIO_PANDOC="/usr/lib/rstudio/bin/pandoc";\
    Rscript -e 'bookdown::render_book("index.Rmd", "bookdown::gitbook")'

# use rstudio pandoc
# this rule sets the environment variable from R using multilines
build_book2:
    Rscript -e "\
    Sys.setenv(RSTUDIO_PANDOC='/usr/lib/rstudio/bin/pandoc');\
    bookdown::render_book('index.Rmd', 'bookdown::gitbook')"
```

Clean up the bookdown project folder

With these two rules I occasionally tidy up and clean up from intermediate files the bookdown project folder:

```
.PHONY: clean
clean: tidy
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.tex -not -name 'preamble.tex' -delete
	$(RM) -rf $(BOOKDOWN_FILES_DIRS)
	$(RM) -rf $(DEFAULT_PUBLISH_BOOK_DIRS)
	if [ -d ${PUBLISH_BOOK_DIR} ]; then rm -rf ${PUBLISH_BOOK_DIR};fi
	if [ -d ${CHECKPOINTS} ]; then rm -rf ${CHECKPOINTS};fi


# delete unwanted files and folders in bookdown folder
.PHONY: tidy
tidy:
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.md -not -name 'README.md' -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name \*-book.html -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.png -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.log -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.rds -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.ckpt -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name \*.nb.html -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name _main.Rmd -delete
	find $(OUTPUT_DIR) -maxdepth 1 -name now.json -delete
```

Chapter 1

Fundamentals

1.1 Library Management

1.1.1 Built-In Libraries

```
import string
import datetime as dt
```

1.1.2 Common External Libraries

```
import numpy as np
import pandas as pd
import datetime as dt

import matplotlib
import matplotlib.pyplot as plt

from plydata import define, query, select, group_by, summarize, arrange, head, rename
import plotnine
from plotnine import *
```

1.1.2.1 numpy

- Large multi-dimensional array and matrices

- High level mathematical funcitons to operate on them
- Efficient array computation, modeled after matlab

- Support vectorized array math functions (built on C, hence faster than python for loop and list)

1.1.2.2 scipy

- Collection of mathematical algorithms and convenience functions built on the numpy extension

- Built upon **numpy**

1.1.2.3 Pandas

- Data manipulation and analysis
- Offer data structures and operations for manipulating numerical tables and time series

- Good for analyzing tabular data

- Use for exploratory data analysis, data pre-processing, statistics and visualization
- Built upon **numpy**

1.1.2.4 scikit-learn

- Machine learning functions

- Built on top of scipy

1.1.2.5 matplotlib

- Data Visualization

1.1.3 Package Management

1.1.4 Conda

1.1.4.1 Conda Environment

```
system("conda info")
```

1.1.4.2 Package Version

```
system("conda list")
```

1.1.4.3 Package Installation

Conda is recommended distribution.

To install from **official** conda channel:

```
conda install <package_name>  # always install latest
conda install <package_name=version_number>

## Example: Install From conda official channel
conda install numpy
conda install scipy
conda install pandas
conda install matpotlib
conda install scikit-learn
conda install seaborn
conda install pip
```

To install from **conda-forge community** channel:

```
conda install -c conda-forge <package_name>
conda install -c conda-forge <package_name=version_number>

## Example: Install From conda community:
conda install -c conda-forge plotnine
```

1.1.5 PIP

PIP is python open repository (not part of conda). Use **pip** if the package is not available in conda.

1.1.5.1 Package Version

```
system("pip list")
```

1.1.5.2 Package Installation

```
pip install <package_name>
## Example: pip install plydata
```

1.2 Everything Is Object

- Every varibales in python are **objects**
- Every variable assginment is **reference based**, that is, each object value is the reference to memory block of data

In the below exmaple, **a, b and c** refer to the **same memory location**:
- Notice when an object assigned to another object, they refer to the same memory location

- When two variable refers to the same value, they refer to the same memory location

```
a = 123
b = 123
c = a
print ('Data of a =',  a,
       '\nData of b =',b,
       '\nData of c =',c,
       '\nID of a = ', id(a),
       '\nID of b = ', id(b),
       '\nID of c = ', id(c)
)
```

```
#:> Data of a = 123
#:> Data of b = 123
#:> Data of c = 123
#:> ID of a =  139663104473568
#:> ID of b =  139663104473568
#:> ID of c =  139663104473568
```

Changing data value (using assignment) changes **the reference**

```
a = 123
b = a
a = 456  # reassignemnt changed a memory reference
         # b memory reference not changed
print ('Data of a =',a,
     '\nData of b =',b,
     '\nID of a = ', id(a),
     '\nID of b = ', id(b)
)
```

```
#:> Data of a = 456
#:> Data of b = 123
#:> ID of a =  139662651314224
#:> ID of b =  139663104473568
```

1.3 Assignment

1.3.1 Multiple Assignment

Assign multiple variable at the same time with same value. Note that all object created using this method refer to the **same memory location**.

```
x = y = 'same mem loc'
print ('x = ', x,
     '\ny = ', y,
```

```
     '\nid(x) = ', id(x),
     '\nid(y) = ', id(y)
)
```

```
#:> x =  same mem loc
#:> y =  same mem loc
#:> id(x) =  139662785737136
#:> id(y) =  139662785737136
```

1.3.2 Augmented Assignment

```
x = 1
y = x + 1
y += 1
print ('y = ', y)
```

```
#:> y =  3
```

1.3.3 Unpacking Assingment

Assign multiple value to multiple variabels at the same time.

```
x,y = 1,3
print (x,y)
```

```
#:> 1 3
```

Chapter 2

Built-in Data Types

2.1 Numbers

Two types of built-in number type, **integer and float**.

2.1.1 Integer

```
n = 123
type (n)
```

```
#:> <class 'int'>
```

2.1.2 Float

```
f = 123.4
type (f)
```

```
#:> <class 'float'>
```

2.1.3 Number Operators

In general, when the operation potentially return **float**, the result is float type. Otherwise it return **integer**.

Division always return float

```
print(4/2)  # return float
```

```
#:> 2.0
```

```
type(4/2)
```

```
#:> <class 'float'>
```

Integer Division by integer return inter. Integer division by float return float.

```
print (8//3,'\n',    # return int
       8//3.2)       # return float
```

```
#:> 2
#:>  2.0
```

Remainder by integer return **integer**.
Remainder by float return **float**

```
print (8%3, '\n',    # return int
       8%3.2)        # return float
```

```
#:> 2
#:>  1.5999999999999996
```

Power return int or float

```
print (2**3)    # return int
```

```
#:> 8
```

```
print (2.1**3)  # return float
```

```
#:> 9.261000000000001
```

```
print (2**3.1)  # return float
```

```
#:> 8.574187700290345
```

2.2 String

String is an object class 'str'. It is an **ordered collection of letters**, an **array** of object type **str**

```
import string
s = 'abcde'
print( '\nvar type  = ', type(s),
       '\nelems     = ',s[0], s[1], s[2],
       '\nlen       = ', len(s),
       '\nelem type = ',type(s[1]))
```

```
#:>
#:> var type  =  <class 'str'>
#:> elems     =  a b c
#:> len       =  5
#:> elem type =  <class 'str'>
```

2.2.1 Constructor

2.2.1.1 Classical Method

`class str(object='')`

```
my_string = str()        ## empty string
```

`class str(object=b'', encoding='utf-8', errors='strict')`

```
my_string = str('abc')
```

2.2.1.2 Shortcut Method

```
my_string = 'abc'
```

2.2.1.3 Multiline Method

```
my_string = '''
This is me.
Yong Keh Soon
'''
print(my_string)
```

```
#:>
#:> This is me.
#:> Yong Keh Soon
```

Note that the variable contain **\n** front and end of the string.

```
my_string
```

```
#:> '\nThis is me.\nYong Keh Soon\n'
```

2.2.1.4 Immutability

- String is **immuatable**. Changing its content will result in **error**

```
s = 'abcde'
print ('s : ', id(s))
#s[1] = 'z'               # immutable, result in error
```

```
#:> s :  139662651357744
```

- Changing the variable completley change the reference (for new object)

```
s = 'efgh'
print ('s : ', id(s))
```

```
#:> s :  139662651305328
```

2.2.2 Class Constants

2.2.2.1 Letters

```
print( 'letters = ', string.ascii_letters,
        '\nlowercase = ',string.ascii_lowercase,
        '\nuppercase = ',string.ascii_uppercase )
```

```
#:> letters =  abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
#:> lowercase =  abcdefghijklmnopqrstuvwxyz
#:> uppercase =  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

2.2.2.2 Digits

```
string.digits
```

```
#:> '0123456789'
```

2.2.2.3 White Spaces

```
string.whitespace
```

```
#:> ' \t\n\r\x0b\x0c'
```

2.2.3 Instance Methods

2.2.3.1 Substitution : `format()`

By Positional

```
print( '{} + {} = {}'.format('a', 'b', 'c'),          # auto sequence
       '\n{0} + {1} = {2}'.format('aa', 'bb', 'cc')) # manual sequence
```

```
#:> a + b = c
#:> aa + bb = cc
```

By Name

```
'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
```

```
#:> 'Coordinates: 37.24N, -115.81W'
```

By Dictionary Name

```
coord = {'latitude': '37.24N', 'longitude': '-115.81W'} ## dictionary key/value
'Coordinates: {latitude}, {longitude}'.format(**coord)
```

```
#:> 'Coordinates: 37.24N, -115.81W'
```

Formatting Number

Float

```
'{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
```

```
#:> '+3.140000; -3.140000'
```

```
'{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
```

```
#:> ' 3.140000; -3.140000'
```

```
'Correct answers: {:.2f}'.format(55676.345345)
```

```
#:> 'Correct answers: 55676.35'
```

Integer, Percentage

```
'{0:,}   {0:.2%}   {0:,.2%}'.format(1234567890.4455)
```

```
#:> '1,234,567,890.4455   123456789044.55%   123,456,789,044.55%'
```

Alignment

```
'{0:<20}   {0:<<20}'.format('left aligned')
```

```
#:> 'left aligned           left aligned<<<<<<<<'
```

```
'{0:>20}  {0:$>20}'.format('right aligned')
```

```
#:> '       right aligned  $$$$$$$right aligned'
```

```
'{:^30}'.format('centered')  # use '*' as a fill char
```

```
#:> '           centered           '
```

2.2.3.2 Substitution : `f-string`

```
my_name = 'Yong Keh Soon'
salary  = 11123.346
f'Hello, {my_name}, your salary is {salary:,.2f} !'
```

```
#:> 'Hello, Yong Keh Soon, your salary is 11,123.35 !'
```

2.2.3.3 Conversion: `upper()` `lower()`

```
'myEXEel.xls'.upper()
```

```
#:> 'MYEXEEL.XLS'
```

```
'myEXEel.xls'.lower()
```

```
#:> 'myexeel.xls'
```

2.2.3.4 `find()` pattern position

`string.find() return position of first occurance. -1 if not found`

```
s='I love karaoke, I know you love it oo'
print (s.find('lov'))
```

```
#:> 2
```

```
print (s.find('kemuning'))
```

```
#:> -1
```

2.2.3.5 `strip()` off blank spaces

```
filename = '  myexce l.   xls   '
filename.strip()
```

```
#:> 'myexce l.   xls'
```

2.2.3.6 List Related: `split()`

Splitting delimeter is specified. Observe the empty spaces were conserved in result array

```
animals = 'a1,a2 ,a3, a4'
animals.split(',')
```

```
#:> ['a1', 'a2 ', 'a3', ' a4']
```

2.2.3.7 List Related: `join()`

```
'-'.join(['1', '2', '3', '4'])
```

```
#:> '1-2-3-4'
```

2.2.3.8 Replacement: `.replace()`

```
string = "geeks for geeks geeks geeks geeks"

# Prints the string by replacing geeks by Geeks
print(string.replace("geeks", "Geeks"))

# Prints the string by replacing only 3 occurrence of Geeks
```

```
#:> Geeks for Geeks Geeks Geeks Geeks
```

```
print(string.replace("geeks", "GeeksforGeeks", 3))
```

```
#:> GeeksforGeeks for GeeksforGeeks GeeksforGeeks geeks geeks
```

2.2.4 Operator

2.2.4.1 % Old Style Substitution

https://docs.python.org/3/library/stdtypes.html#old-string-formatting

```
my_name = 'Yong Keh Soon'
salary  = 11123.346
'Hello, %s, your salary is %.2f !' %(my_name, salary)
```

```
#:> 'Hello, Yong Keh Soon, your salary is 11123.35 !'
```

2.2.4.2 + Concatenation

```
'this is ' + 'awesome'
```

```
#:> 'this is awesome'
```

2.2.4.3 in matching

For single string, **partial match**

```
print( 'abc' in '123abcdefg' )
```

```
#:> True
```

For list of strings, **exact match** (even though only one element in list).
For partial match, workaround is to **convert list to single string**

```
print( 'abc' in ['abcdefg'],             # false
       'abc' in ['abcdefg','123'],       # fakse
       'abc' in ['123','abc','def'],     # true
       'abc' in str(['123','abcdefg']))  # true
```

```
#:> False False True True
```

2.2.4.4 Comparitor

Comparitor compares the memory address.

```
a='abc'
b='abc'
print('id(a) = ', id(a),
      '\nid(b) = ', id(b),
      '\na == b  ', a==b)
```

```
#:> id(a) =  139663100038384
#:> id(b) =  139663100038384
#:> a == b   True
```

2.2.5 Iterations

```
string[start:end:step]  # default start:0, end:last, step:1
```

If step is negative (reverse), end value must be lower than start value

```
s = 'abcdefghijk'
print (s[0])       # first later
```

```
#:> a
```

```
print (s[:3])      # first 3 letters
```

```
#:> abc
```

```
print (s[2:8 :2])  # stepping
```

```
#:> ceg
```

```
print (s[-1])      # last letter
```

```
#:> k
```

```
print (s[-3:])     # last three letters
```

```
#:> ijk
```

```
print (s[:   :-1]) # reverse everything
```

```
#:> kjihgfedcba
```

```
print (s[8:2 :-1])
```

```
#:> ihgfed
```

```
print (s[8:2])     # return NOTHING
```

2.3 Boolean

```
b = False

if (b):
    print ('It is true')
else:
    print ('It is fake')
```

```
#:> It is fake
```

2.3.1 What is Considered False ?

Everything below are false, **anything else are true**

```
print ( bool(0),       # zero
        bool(None),  # none
        bool(''),    # empty string
        bool([]),    # empty list
        bool(()),    # empty tupple
        bool(False), # False
        bool(2-2))    # expression that return any value above
```

```
#:> False False False False False False False
```

2.3.2 and operator

BEWARE !

- **and** can return different data types
- If evaluated result is **True**, the last **True Value** is returned (because python need to evaluate up to the last value)
- If evaluated result is **False**, the first **False Value** will be returned (because python return it immediately when detecting False value)

```
print (123 and 2 and 1,
       123 and [] and 2)
```

```
#:> 1 []
```

2.3.3 not operator

```
not (True)
```

```
#:> False
```

```
not (True or False)
```

```
#:> False
```

```
not (False)
```

```
#:> True
```

```
not (True and False)
```

```
#:> True
```

```
~(False)
```

```
#:> -1
```

2.3.4 `or` operator

- **or** can return different data type
- If evaluated result is True, first **True Value** will be returned (right hand side value **need not be evaluated**)
- If evaluated result is False, last **Fasle Value** will be returned (need to evalute all items before concluding False)

```
print (1 or 2)
```

```
#:> 1
```

```
print (0 or 1 or 1)
```

```
#:> 1
```

```
print (0 or () or [])
```

```
#:> []
```

2.4 None

2.4.1 None is an Object

- None is a Python **object NonType**
- Any operation to None object will result in **error**
- For array data with None elements, verification is required to check through iteration to determine if the item is not None. It is very computaionaly heavy

```
type(None)
```

```
#:> <class 'NoneType'>
```

```
t1 = np.array([1, 2, 3, 4, 5])
t2= np.array([1, 2, 3, None, 4, 5])
print( t1.dtype  , '\n\n',    # it's an object
       t2.dtype)
```

```
#:> int64
#:>
#:>  object
```

2.4.2 Comparing None

Not Prefered Method

```
null_variable = None
print( null_variable == None )
```

```
#:> True
```

Prefered

```
print( null_variable is None )
```

```
#:> True
```

```
print( null_variable is not None )
```

```
#:> False
```

2.4.3 Operation on None

Any operator (except `is`) on None results in error.

```
None & None
```

```
#:> Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported operand typ
#:>
#:> Detailed traceback:
#:>   File "<string>", line 1, in <module>
```

Chapter 3

Built-In Data Structure

3.1 Tuple

Tuple is an **immutable list**. Any attempt to change/update tuple will return error. It can contain **different types** of object.

Benefits of tuple against List are: - **Faster** than list - **Protects** your data against accidental change - Can be used as key in dictionaries, list can't

3.1.1 Creating

3.1.1.1 Constructor

```
# mylist = [1,2,3]
# print(tuple(mylist))
```

3.1.1.2 Assignment

With or Without `()`

This is a formal syntax for defining tuple, items inside ( ) notation. Assignment although works without `()`, it is not recommended.

```
t1 = (1,2,3,'o','apple')
t2 = 1,2,3,'o','apple'

print(type(t1), type(t2))
```

```
#:> <class 'tuple'> <class 'tuple'>
```

3.1.2 Accessing

```
print( t[1], t[1:3] )
```

3.1.3 Duplicating

Use normal assignment `=` to duplicate. Reference of the memory address is copied. Data is actually not duplicated in memory.

```
original = (1,2,3,4,5)
copy_test = original
print(original)
```

```
#:> (1, 2, 3, 4, 5)
```

```
print(copy_test)
```

```
#:> (1, 2, 3, 4, 5)
```

The copy and original has the same memory location.

```
print('Original ID: ', id(original))
```

```
#:> Original ID:  139662651247984
```

```
print('Copy ID:     ', id(copy_test))
```

```
#:> Copy ID:      139662651247984
```

3.2 List

- List is a collection of **ordered** items, where the items **can be different data types**
- You can pack list of items by placing them into []
- List is mutable

3.2.1 Creating List

3.2.1.1 Empty List

```
empty = []      # literal assignment method
empty = list()  # constructor method
print (empty)
```

```
#:> []
```

3.2.1.2 Literal Assignment

- **Multiple data types** is allowed in a list

```
[123,'abc',456, None]
```

```
#:> [123, 'abc', 456, None]
```

Constructor

- Note that **list(string)** will split the string into letters

```
list('hello')
```

```
#:> ['h', 'e', 'l', 'l', 'o']
```

3.2.2 Accessing Items

Access specific index number

```
food = ['bread', 'noodle', 'rice', 'biscuit','jelly','cake']
print (food[2])  # 3rd item
```

```
#:> rice
```

```
print (food[-1]) # last item
```

```
#:> cake
```

Access range of indexes

```
print (food[:4])     # first 3 items
```

```
#:> ['bread', 'noodle', 'rice', 'biscuit']
```

```
print (food[-3:])    # last 3 items
```

```
#:> ['biscuit', 'jelly', 'cake']
```

```
print (food[1:5])    # item 1 to 4
```

```
#:> ['noodle', 'rice', 'biscuit', 'jelly']
```

```
print (food[5:2:-1]) # item 3 to 5, reverse order
```

```
#:> ['cake', 'jelly', 'biscuit']
```

```
print (food[::-1])   # reverse order
```

```
#:> ['cake', 'jelly', 'biscuit', 'rice', 'noodle', 'bread']
```

3.2.3 Methods

3.2.3.1 Remove Item(s)

Removal of non-existance item will result in error

Search and remove first matching item

```
food = list(['bread', 'noodle', 'rice', 'biscuit','jelly','cake','noodle'])
food.remove('noodle')
print (food)
```

```
#:> ['bread', 'rice', 'biscuit', 'jelly', 'cake', 'noodle']
```

Remove last item

```
food.pop()
```

```
#:> 'noodle'
```

```
print (food)
```

```
#:> ['bread', 'rice', 'biscuit', 'jelly', 'cake']
```

Remove item at specific position

```
food.pop(1)  # counter start from 0
```

```
#:> 'rice'
```

```
print(food)
```

```
#:> ['bread', 'biscuit', 'jelly', 'cake']
```

```
food.remove('jelly')
print(food)
```

```
#:> ['bread', 'biscuit', 'cake']
```

3.2.3.2 Appending Item (s)

Append One Item

```
food.append('jelly')
print (food)
```

```
#:> ['bread', 'biscuit', 'cake', 'jelly']
```

Append Multiple Items **`extend()`** will expand the list/tupple argument and append as multiple items

```
food.extend(['nand','puff'])
print (food)
```

```
#:> ['bread', 'biscuit', 'cake', 'jelly', 'nand', 'puff']
```

3.2.3.3 Other Methods

Reversing the order of the items

```
food.reverse()
food
```

```
#:> ['puff', 'nand', 'jelly', 'cake', 'biscuit', 'bread']
```

Locating the Index Number of An Item

```
food.index('biscuit')
```

```
#:> 4
```

Count occurance

```
test = ['a','a','a','b','c']
test.count('a')
```

```
#:> 3
```

Sorting The Order of Items

```
food.sort()
print (food)
```

```
#:> ['biscuit', 'bread', 'cake', 'jelly', 'nand', 'puff']
```

3.2.4 Operator

3.2.4.1 Concatenation

Concatenating Lists

Two lists can be concatenanted using '+' operator.

```
['dog','cat','horse'] + ['elephant','tiger'] + ['sheep']
```

```
#:> ['dog', 'cat', 'horse', 'elephant', 'tiger', 'sheep']
```

3.2.5 List is Mutable

The reference of list variable won't change after adding/removing its item

```
food = ['cake','jelly','roti','noodle']
print ('food : ',id(food))
```

```
#:> food :  139662651351600
```

```
food += ['salad','chicken']
print ('food : ',id(food))
```

```
#:> food :  139662651351600
```

A function is actually an **object**, which reference never change, hence **mutable**

```
def spam (elem, some_list=['a','b']):
    some_list.append(elem)
    return some_list

print (spam(1,['x']))
```

```
#:> ['x', 1]
```

```
print (spam(2)) ## second parameter is not passed
```

```
#:> ['a', 'b', 2]
```

```
print (spam(3)) ##  notice the default was remembered
```

```
#:> ['a', 'b', 2, 3]
```

3.2.6 Duplicate or Reference

Use = : It just copy the refernce. IDs are similar

```
original = [1,2,3,4,5]
copy_test = original
print('Original ID: ', id(original))
```

```
#:> Original ID:  139662651327904
```

```
print('Copy ID:     ', id(copy_test))
```

```
#:> Copy ID:      139662651327904
```

```
original[0]=999   ## change original
print(original)
```

```
#:> [999, 2, 3, 4, 5]
```

```
print(copy_test)  ## copy affected
```

```
#:> [999, 2, 3, 4, 5]
```

Duplicate A List Object with `copy()`. Resulting IDs are different

```
original = [1,2,3,4,5]
copy_test = original.copy()
print(original)
```

```
#:> [1, 2, 3, 4, 5]
```

```
print(copy_test)
```

```
#:> [1, 2, 3, 4, 5]
```

```
print('Original ID: ', id(original))
```

```
#:> Original ID:  139662651327104
```

```
print('Copy ID:     ', id(copy_test))
```

```
#:> Copy ID:      139662651298272
```

```
original[0] = 999  ## change original
print(original)
```

```
#:> [999, 2, 3, 4, 5]
```

```
print(copy_test)   ## copy not affected
```

```
#:> [1, 2, 3, 4, 5]
```

Passing To Function As Reference

```
def func(x):
    print (x)
    print('ID in Function:      ', id(x))
    x.append(6)    ## modify the refrence

my_list = [1,2,3,4,5]
print('ID outside Function: ', id(my_list))
```

```
#:> ID outside Function:  139662651326368
```

```
func(my_list)  ## call the function, pass the reference
```

```
#:> [1, 2, 3, 4, 5]
#:> ID in Function:       139662651326368
```

```
print(my_list) ## content was altered
```

```
#:> [1, 2, 3, 4, 5, 6]
```

3.2.7 List Is Iterable

3.2.7.1 For Loop

```
s = ['abc','abcd','bcde','bcdee','cdefg']
for x in s:
    if 'abc' in x:
        print (x)
```

```
#:> abc
#:> abcd
```

3.2.7.2 List Comprehension

This code below is a shorform method of **for loop and if**.

```
old_list = ['abc','abcd','bcde','bcdee','cdefg']
[x for x in old_list if 'abc' in x]
```

```
#:> ['abc', 'abcd']
```

Compare to traditional version of code below:

```
new_list = []
old_list = ['abc','abcd','bcde','bcdee','cdefg']
for x in old_list:
    if 'abc' in x:
        new_list.append(x)

print( new_list )
```

```
#:> ['abc', 'abcd']
```

3.2.8 Conversion

Convert mutable list to immutable **tuple** with **`tuple()`**

```
original = [1,2,3]
original_tuple = tuple(original)
print( id(original),
       id(original_tuple))
```

```
#:> 139662651300192 139662651378016
```

3.2.9 Built-In Functions Applicable To List

Number of Elements

```
len(food)
```

```
#:> 6
```

Max Value

```
test = [1,2,3,5,5,3,2,1]
m = max(test)
test.index(m)  ## only first occurance is found
```

```
#:> 3
```

3.3 Dictionaries

Dictionary is a list of index-value items.

3.3.1 Creating dict

3.3.1.1 From Literals

Simple Dictionary

```
animal_counts = { 'cats' : 2, 'dogs' : 5, 'horses':4}
print (animal_counts)
```

```
#:> {'cats': 2, 'dogs': 5, 'horses': 4}
```

```
print( type(animal_counts) )
```

```
#:> <class 'dict'>
```

Dictionary with list

```
animal_names = {'cats':   ['Walter','Ra'],
                'dogs':   ['Jim','Roy','John','Lucky','Row'],
                'horses': ['Sax','Jack','Ann','Jeep']
               }
animal_names
```

```
#:> {'cats': ['Walter', 'Ra'], 'dogs': ['Jim', 'Roy', 'John', 'Lucky', 'Row'], 'horses': ['Sax',
```

3.3.1.2 From Variables

```
cat_names = ['Walter','Ra','Jim']
dog_names = ['Jim','Roy','John','Lucky','Row']
horse_names= ['Sax','Jack','Ann','Jeep']
animal_names = {'cats': cat_names, 'dogs': dog_names, 'horses': horse_names}
animal_names
```

```
#:> {'cats': ['Walter', 'Ra', 'Jim'], 'dogs': ['Jim', 'Roy', 'John', 'Lucky', 'Row'], 'horses': [
```

3.3.2 Accessing dict

3.3.2.1 Get All Keys

```
print (animal_names.keys())
```

```
#:> dict_keys(['cats', 'dogs', 'horses'])
```

```
print (sorted(animal_names.keys()))
```

```
#:> ['cats', 'dogs', 'horses']
```

3.3.2.2 Get All Values

```
print (animal_names.values())
```

```
#:> dict_values([['Walter', 'Ra', 'Jim'], ['Jim', 'Roy', 'John', 'Lucky', 'Row'], ['Sa
```

```
print (sorted(animal_names.values()))
```

```
#:> [['Jim', 'Roy', 'John', 'Lucky', 'Row'], ['Sax', 'Jack', 'Ann', 'Jeep'], ['Walter'
```

3.3.2.3 Access value with Specific Key

Use **`[ key ]`** notation. However, this will return **Error** if key does not exist

```
animal_names['dogs']
```

```
#:> ['Jim', 'Roy', 'John', 'Lucky', 'Row']
```

Use **`get( key )`** notation. will return None if key does not exist

```
print (animal_counts.get('cow'))
```

```
#:> None
```

3.3.3 Dict Is Mutable

3.3.3.1 Update/Append

Use [**key**] notation to **update** o **append** the content of element.

```
animal_names['dogs'] = ['Ali','Abu','Bakar']
animal_names
```

```
#:> {'cats': ['Walter', 'Ra', 'Jim'], 'dogs': ['Ali', 'Abu', 'Bakar'], 'horses': ['Sax
```

Use **`clear()`** to erase all elements

```
animal_names.clear()
```

3.3.4 Iterating Elements

Loop through **`.items()`**

```
animal_dict = { 'cats' : 2, 'dogs' : 5, 'horses':4}

for key,val in animal_dict.items():
  print( key, val )
```

```
#:> cats 2
#:> dogs 5
#:> horses 4
```

3.4 Sets

Set is **unordered** collection of **unique items**. Set is **mutable**

3.4.1 Creation

Set can be declared with `{}`, unlike list creation uses '[]'.

```
myset = {'a','b','c','d','a','b','e','f','g'}
print (myset) # notice no repetition values
```

```
#:> {'g', 'c', 'e', 'a', 'd', 'f', 'b'}
```

Set can be created from list, and then converted back to list

```
mylist = ['a','b','c','d','a','b','e','f','g']
myset = set(mylist)
my_unique_list = list(myset)
print (
  'Original List       : ', mylist,
  '\nConvert to set      : ', myset,
  '\nConvert back to list: ', my_unique_list) # notice no repetition values
```

```
#:> Original List       :  ['a', 'b', 'c', 'd', 'a', 'b', 'e', 'f', 'g']
#:> Convert to set      :  {'g', 'c', 'e', 'a', 'd', 'f', 'b'}
#:> Convert back to list:  ['g', 'c', 'e', 'a', 'd', 'f', 'b']
```

3.4.2 Membership Test

```
print ('a' in myset)      # is member ?
```

```
#:> True
```

```
print ('f' not in myset)  # is not member ?
```

```
#:> False
```

3.4.3 Subset Test

Subset Test : <=
Proper Subset Test : <

```
mysubset = {'d','g'}
mysubset <= myset
```

```
#:> True
```

Proper Subset test that the master set **contain at least one element** which is not in the subset

```
mysubset = {'b','a','d','c','e','f','g'}
print ('Is Subset : ', mysubset <= myset)
```

```
#:> Is Subset :  True
```

```
print ('Is Proper Subet : ', mysubset < myset)
```

```
#:> Is Proper Subet :  False
```

3.4.4 Union using |

```
{'a','b','c'} | {'a','e','f'}
```

```
#:> {'a', 'c', 'f', 'b', 'e'}
```

3.4.5 Intersection using &

Any elments that exist in both left and right set

```
{'a','b','c','d'} & {'c','d','e','f'}
```

```
#:> {'c', 'd'}
```

3.4.6 Difference using -

Remove **right** from **left**

```
{'a','b','c','d'} - {'c','d','e','f'}
```

```
#:> {'a', 'b'}
```

3.5 range

range(X) generates sequence of integer object

```
range (lower_bound, upper_bound, step_size)
# lower bound is optional, default = 0
# upper bound is not included in result
# step is optional, default = 1
```

Use list() to convert in order to view actual sequence of data

```
r = range(10)     # default lower bound =0, step =1
print (type (r))
```

```
#:> <class 'range'>
```

```
print (r)
```

```
#:> range(0, 10)
```

```
print (list(r))
```

```
#:> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

More Examples

```
print (list(range(2,8)))    # step not specified, default 1
```

```
#:> [2, 3, 4, 5, 6, 7]
```

```
print ('Odds Number : ' , list(range(1,10,2))) # generate odds number
```

```
#:> Odds Number :  [1, 3, 5, 7, 9]
```

Chapter 4

Control and Loops

4.1 If Statement

4.1.1 Multiline If.. Statements

```
price = 102
if price <100:
    print ('buy')
elif price < 110:
    print ('hold')
elif price < 120:
    print ('think about it')
else:
    print ('sell')
```

```
#:> hold
```

```
print('end of programming')
```

```
#:> end of programming
```

4.1.2 Single Line If .. Statement

4.1.2.1 if ... In One Statement

```
price = 70
if price<80: print('buy')
```

```
#:> buy
```

4.1.2.2 Ternary Statemnt

This statement return a value with simple condition

```
price = 85
'buy' if (price<80) else 'dont buy'
```

```
#:> 'dont buy'
```

4.2 For Loops

4.2.1 For .. Else Construct

`else` is only executed when the for loop **completed all cycles**

```
mylist = [1,2,3,4,5]

for i in mylist:
  print (i)
else:
  print('Hooray, the loop is completed successfully')
```

```
#:> 1
#:> 2
#:> 3
#:> 4
#:> 5
#:> Hooray, the loop is completed successfully
```

In below exmaple, for loop encountered `break`, hence the `else` section is not executed.

```
for i in mylist:
  if i < 4:
    print (i)
  else:
    print('Oops, I am breaking out half way in the loop')
    break
else:
  print('Hooray, the loop is completed successfully')
```

```
#:> 1
#:> 2
#:> 3
#:> Oops, I am breaking out half way in the loop
```

4.2.2 Loop thorugh 'range'

```
for i in range (1,10,2):
    print ('Odds Number : ',i)
```

```
#:> Odds Number :  1
#:> Odds Number :  3
#:> Odds Number :  5
#:> Odds Number :  7
#:> Odds Number :  9
```

4.2.3 Loop through 'list'

4.2.3.1 Standard For Loop

```
letters = ['a','b','c','d']
for e in letters:
    print ('Letter : ',e)
```

```
#:> Letter :  a
#:> Letter :  b
#:> Letter :  c
#:> Letter :  d
```

4.2.3.2 List Comprehension

Iterate through existing list, and **build new list** based on condition
`new_list = [expression(i) for i in old_list]`

```
s = ['abc','abcd','bcde','bcdee','cdefg']
[x.upper() for x in s]
```

```
#:> ['ABC', 'ABCD', 'BCDE', 'BCDEE', 'CDEFG']
```

Extend list comprehension can be extended with `if` condition**
`new_list = [expression(i) for i in old_list if filter(i)]`

```
old_list    = ['abc','abcd','bcde','bcdee','cdefg']
matching = [ x.upper() for x in old_list if 'bcd' in x ]
print( matching )
```

```
#:> ['ABCD', 'BCDE', 'BCDEE']
```

4.2.4 Loop Through 'Dictionary'

Looping through dict will picup **key**

```
d = {"x": 1, "y": 2}
for key in d:
    print (key, d[key])
```

```
#:> x 1
#:> y 2
```

4.3 Generators

- Generator is lazy, produce items only if asked for, hence more memory efficient
- Generator is **function** with 'yield' instead of 'return'
- Generator contains one or more yields statement
- When called, it returns an object (iterator) but **does not start execution** immediately
- Methods like **iter**() and **next**() are implemented automatically. So we can iterate through the items using **next()**
- Once the function yields, the **function is paused** and the control is transferred to the caller
- Local variables and their states are **remembered** between successive calls
- Finally, when the function **terminates**, **StopIteration** is raised automatically on further calls

4.3.1 Basic Generator Function

Below example give clear understanding of how generator works

```
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
```

```
    yield n
```

```
a = my_gen()
type(a)
```

```
#:> <class 'generator'>
```

```
next(a)
```

```
#:> This is printed first
#:> 1
```

```
next(a)
```

```
#:> This is printed second
#:> 2
```

4.3.2 Useful Generator Fuction

Generator is only useful when it uses **for-loop** - for-loop within generator - for-loop to iterate through a generator

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]
```

```
for c in rev_str("hello"):
     print(c)
```

```
#:> o
#:> l
#:> l
#:> e
#:> h
```

4.3.3 Generator Expression

Use () to create an annonymous generator function

```
my_list = [1, 3, 6, 10]
a = (x**2 for x in my_list)
```

```
next(a)
```

```
#:> 1
```

```
next(a)
```

```
#:> 9
```

```
sum(a) # sum the power of 6,10
```

```
#:> 136
```

4.3.4 Compare to Iterator Class

```
class PowTwo:
    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

Obviously, Generator is more concise and cleaner

```
def PowTwoGen(max = 0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Chapter 5

Library and Functions

Library are group of functions

5.1 Package Source

5.1.1 Conda

- Package manager for any language
- Install binaries

5.1.2 PIP

- Package manager python only
- Compile from source
- Stands for Pip Installs Packages
- Python's officially-sanctioned package manager, and is most commonly used to install packages published on the **Python Package Index (PyPI)**
- Both pip and PyPI are governed and supported by the Python Packaging Authority (PyPA).

5.2 Importing Library

There are two methods to import library functions:

Standalone Namespace

```
- import <libName>                        # access function through: libName.functionNa
- import <libName> as <shortName>         # access function through: shortName.functio
```

Global Namespace

```
- from   <libName> import *               # all functions available at global namespace
- from   <libName> import <functionName>  # access function through: functionName
- from   <libName> import <functionName> as <shortFunctionName>  # access function thro
```

5.2.1 Import Entire Library

5.2.1.1 Import Into Standalone Namespace

```
import math
math.sqrt(9)
```

```
#:> 3.0
```

Use **as** for aliasing library name. This is useful if you have conflicting library name

```
import math as m
m.sqrt(9)
```

```
#:> 3.0
```

5.2.1.2 Import Into Global Name Space

All functions in the library accessible through global namespace

```
from <libName> import *
```

5.2.2 Import Specific Function

```
from math import sqrt
print (sqrt(9))
```

```
#:> 3.0
```

Use **as** for aliasing function name

```
from math import sqrt as sq
print (sq(9))
```

```
#:> 3.0
```

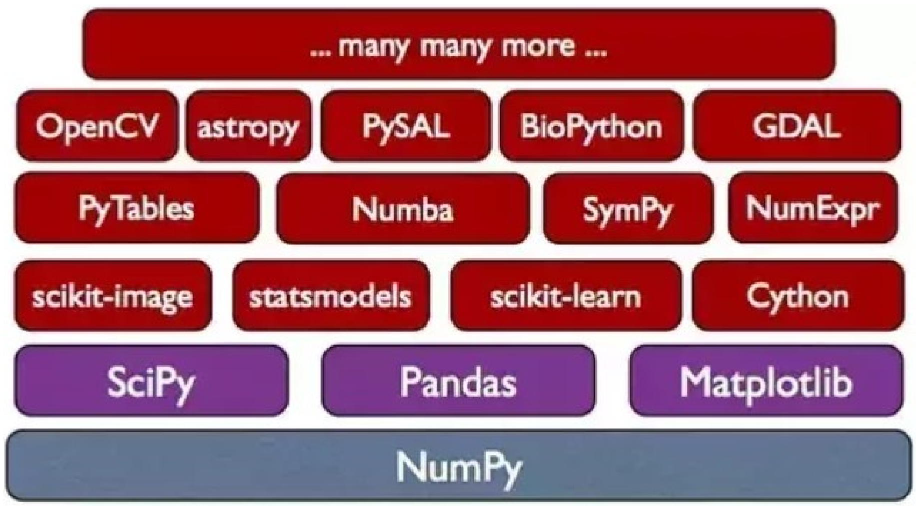


Figure 5.1: alt text

5.2.3 Machine Learning Packages

5.3 Define Function

5.3.1 Function Arguments

By default, arguments are assigned to function left to right

```
def myfun(x,y):
    print ('x:',x)
    print ('y:',y)

myfun(5,8)
```

```
#:> x: 5
#:> y: 8
```

However, you can also specify the argument assigment during function call

```
myfun (y=8,x=5)
```

```
#:> x: 5
#:> y: 8
```

Function can have **default argement value**

```
def myfun(x=1,y=1):  # default argument value is 1
    print ('x:',x)
    print ('y:',y)
```

```
myfun(5)  # pass only one argument
```

```
#:> x: 5
#:> y: 1
```

5.3.2 List Within Function

Consider a function is an object, its variable (some_list) is immutable and hence its reference won't change, even data changes

```
def spam (elem, some_list=[]):
    some_list.append(elem)
    return some_list

print (spam(1))
```

```
#:> [1]
```

```
print (spam(2))
```

```
#:> [1, 2]
```

```
print (spam(3))
```

```
#:> [1, 2, 3]
```

5.3.3 Return Statement

```
def bigger(x,y):
    if (x>y):
        return x
    else:
        return y

print (bigger(5,8))
```

```
#:> 8
```

5.3.4 No Return Statement

if no **return** statement, python return **None**

```
def dummy():
    print ('This is a dummy function, return no value')

dummy()
```

```
#:> This is a dummy function, return no value
```

5.3.5 Return Multiple Value

Multiple value is returned as **tuple**. Use multiple assignment to assign to multiple variable

```
def minmax(x,y,z):
    return min(x,y,z), max(x,y,z)

a,b = minmax(7,8,9)     # multiple assignment
c   = minmax(7,8,9)     # tuple

print (a,b)
```

```
#:> 7 9
```

```
print (c)
```

```
#:> (7, 9)
```

5.3.6 Passing Function as Argument

You can pass a function name as an argument to a function

```
def myfun(x,y,f):
    f(x,y)

myfun('hello',54,print)
```

```
#:> hello 54
```

5.3.7 Arguments

args is a **tuple**

5.3.7.1 Example 1

Error example, too many parameters passed over to function

5.3.7.2 Example 2

First argument goes to x, remaining goes to args as tuple

```
def myfun(x,*args):
    print (x)
    print (args)     #tuple

myfun(1,2,3,4,5,'abc')
```

```
#:> 1
#:> (2, 3, 4, 5, 'abc')
```

5.3.7.3 Example 3

First argument goes to x, second argument goest to y, remaining goes to args

```
def myfun(x,y,*args):
    print (x)
    print (y)
    print (args)     #tuple

myfun(1,2,3)
```

```
#:> 1
#:> 2
#:> (3,)
```

5.3.7.4 Example 4

```
def myfun(x,*args, y=9):
    print (x)
    print (y)
    print (args)     #tuple

myfun(1,2,3,4,5)
```

```
#:> 1
#:> 9
#:> (2, 3, 4, 5)
```

5.3.7.5 Example 5

All goes to args

```
def myfun(*args):
    print (args)     #tuple

myfun(1,2,3,4,5)
```

```
#:> (1, 2, 3, 4, 5)
```

5.3.7.6 Example 6 Empty args

```
def myfun(x,y,*args):
    print (x)
    print (y)
    print (args)

myfun(1,2)
```

```
#:> 1
#:> 2
#:> ()
```

5.3.8 keyword arguments

kwargs is a **dictionary**

5.3.8.1 Example 1

```
def foo(**kwargs):
    print(kwargs)

foo(a=1,b=2,c=3)
```

```
#:> {'a': 1, 'b': 2, 'c': 3}
```

5.3.8.2 Example 2

```
def foo(x,**kwargs):
    print(x)
    print(kwargs)

foo(9,a=1,b=2,c=3)
```

```
#:> 9
#:> {'a': 1, 'b': 2, 'c': 3}
```

```
foo(9) #empty dictionary
```

```
#:> 9
#:> {}
```

5.3.8.3 Example 3

```
def foo(a,b,c,d=1):
    print(a)
    print(b)
    print(c)
    print(d)

foo(**{"a":2,"b":3,"c":4})
```

```
#:> 2
#:> 3
#:> 4
#:> 1
```

5.3.9 Mixing *args, **kwargs

Always put args **before** kwargs

5.3.9.1 Example 1

```
def foo(x,y=1,**kwargs):
    print (x)
    print (y)
    print (kwargs)

foo(1,2,c=3,d=4)
```

```
#:> 1
#:> 2
#:> {'c': 3, 'd': 4}
```

5.3.9.2 Example 2

```
def foo(x,y=2,*args,**kwargs):
    print (x)
    print (y)
    print (args)
    print (kwargs)

foo(1,2,3,4,5,c=6,d=7)
```

```
#:> 1
#:> 2
#:> (3, 4, 5)
#:> {'c': 6, 'd': 7}
```

Chapter 6

Exception Handling

The try statement works as follows:
- First, the try clause (the statement(s) between the try and except keywords) is executed
- If no exception occurs, the except clause is skipped and execution of the try statement is finished
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above

A try statement may have more than one except clause, to specify handlers for different exceptions.

6.1 Catching Error

Different exception object has different attributes.

```
try:
  a = 1 + 'a'

## known error
except TypeError as err:
  print('I know this error !!!!',
        '\n Error: ', err,
        '\n Args:  ', err.args,
        '\n Type:  ', type(err))
```

```
## all other error
except Exception as err:
  print( 'Error: ', err,
         '\nArgs:  ', err.args,
         '\nType:  ', type(err))
```

```
#:> I know this error !!!!
#:>  Error:  unsupported operand type(s) for +: 'int' and 'str'
#:>  Args:   ("unsupported operand type(s) for +: 'int' and 'str'",)
#:>  Type:   <class 'TypeError'>
```

6.2 Custom Exception

```
try:
  raise Exception('bloody', 'hell')  #simulate exception
except Exception as err:
  print( 'Error: ', err,
         '\nArgs:  ', err.args,
         '\nType:  ', type(err))
```

```
#:> Error:  ('bloody', 'hell')
#:> Args:   ('bloody', 'hell')
#:> Type:   <class 'Exception'>
```

Chapter 7

Object Oriented Programming

7.1 Defining Class

- Every function within a class **must have** at least one parameter - **self**
- Use **init** as the constructor function. **init** is optional

```
class Person:
  wallet = 0  #
  def __init__(self, myname,money=0):   # constructor
      self.name = myname
      self.wallet=money
      print('I\'m in Person Constructor: {}'.format(myname))
  def say_hi(self):
      print('Hello, my name is : ', self.name)
  def say_bye(self):
      print('Goodbye', Person.ID)
  def take(self,amount):
      self.wallet+=amount
  def balance(self):
      print('Wallet Balance:',self.wallet)
  def MakeCry(self):
      self.Cry()

class Kid(Person):
  def __init__(self, myname, money=0):
      print('I\'m in Kid Constructor: {}'.format(myname))
      super().__init__(myname=myname, money=money)
```

```
    def Cry(self):
        print('Kid is crying')
```

7.2 Constructor

```
p1 = Person('Yong')
```

```
#:> I'm in Person Constructor: Yong
```

```
p2 = Person('Gan',200)
```

```
#:> I'm in Person Constructor: Gan
```

```
p3 = Kid('Jolin',50)
```

```
#:> I'm in Kid Constructor: Jolin
#:> I'm in Person Constructor: Jolin
```

7.3 Calling Method

```
p1.say_hi()
```

```
#:> Hello, my name is :  Yong
```

```
p1.balance()
```

```
#:> Wallet Balance: 0
```

```
p3.Cry()
```

```
#:> Kid is crying
```

```
p3.MakeCry()
```

```
#:> Kid is crying
```

```
p2.say_hi()
```

```
#:> Hello, my name is :  Gan
```

```
p2.balance()
```

```
#:> Wallet Balance: 200
```

7.4 Getting Property

```
p1.wallet
```

```
#:> 0
```

```
p2.wallet
```

```
#:> 200
```

7.5 Setting Property

```
p1.wallet = 900
p1.wallet
```

```
#:> 900
```

Chapter 8

Decorator

8.1 Definition

- **Decorator** is a function that accept callable as the **only argument**
- The main purpose of decarator is to **enhance** the program of the decorated function
- It returns a **callable**

8.2 Examples

8.2.1 Example 1 - Plain decorator function

- Many times, it is useful to register a function elsewhere - for example, registering a task in a task runner, or a functin with signal handler
- **register** is a decarator, it accept **decorated** as the only argument
- foo() and bar() are the **decorated function** of **register**

```
registry = []

def register(decorated):
    registry.append(decorated)
    return decorated

@register
def foo():
    return 3

@register
def bar():
    return 5
```

```
registry
```

```
#:> [<function foo at 0x7f05beb86320>, <function bar at 0x7f05beb86290>]
```

```
registry[0]()
```

```
#:> 3
```

```
registry[1]()
```

```
#:> 5
```

8.2.2 Example 2 - Decorator with Class

- Extending the use case above
- register is the **decarator**, it has only one argument

```
class Registry(object):
    def __init__(self):
        self._functions = []
    def register(self,decorated):
        self._functions.append(decorated)
        return decorated
    def run_all(self,*args,**kwargs):
        return_values = []
        for func in self._functions:
            return_values.append(func(*args,**kwargs))
        return return_values
```

The decorator will decorate two functions, for both object **a** and **b**

```
a = Registry()
b = Registry()

@a.register
def foo(x=3):
    return x

@b.register
def bar(x=5):
    return x

@a.register
@b.register
def bax(x=7):
    return x
```

Observe the result

```
print (a._functions)
```

```
#:> [<function foo at 0x7f05beba3560>, <function bax at 0x7f05beba38c0>]
```

```
print (b._functions)
```

```
#:> [<function bar at 0x7f05beba3710>, <function bax at 0x7f05beba38c0>]
```

```
print (a.run_all())
```

```
#:> [3, 7]
```

```
print (b.run_all())
```

```
#:> [5, 7]
```

```
print ( a.run_all(x=9) )
```

```
#:> [9, 9]
```

```
print ( b.run_all(x=9) )
```

```
#:> [9, 9]
```

Chapter 9

datetime Standard Library

This is a built-in library by Python. There is no need to install this library.

9.1 ISO8601

https://en.wikipedia.org/wiki/ISO_8601#Time_zone_designators

9.1.1 Date Time

`UTC: "2007-04-05T14:30Z" #notice Z GMT+8: "2007-04-05T12:30+08:00 #notice +08:00 GMT+8: "2007-04-05T12:30+0800 #notice +0800 GMT+8: "2007-04-05T12:30+08 #notice +08`

9.1.2 Date

2019-02-04 #notice no timezone available

9.2 Module Import

```
from datetime import date     # module for date object
from datetime import time     # module for time object
from datetime import datetime # module for datetime object
from datetime import timedelta
```

9.3 Class

datetime library contain **three class of objects**:
- **date** (year,month,day)

- **time** (hour,minute,second)
- **datetime** (year,month,day,hour,minute,second)
- **timedelta**: duration between two datetime or date object

9.4 date

9.4.1 Constructor

```
print( date(2000,1,1) )
```

```
#:> 2000-01-01
```

```
print( date(year=2000,month=1,day=1) )
```

```
#:> 2000-01-01
```

```
print( type(date(year=2000,month=1,day=1)))
```

```
#:> <class 'datetime.date'>
```

9.4.2 Class Method

9.4.2.1 today

This is **local date** (not UTC)

```
date.today()
```

```
#:> datetime.date(2020, 12, 27)
```

```
print( date.today() )
```

```
#:> 2020-12-27
```

9.4.2.2 Convert From ISO fromisoformat

`strptime` is **not available for date** conversion. It is only for datetime conversion

```
date.fromisoformat('2011-11-11')
```

```
#:> datetime.date(2011, 11, 11)
```

To convert **non-iso format** date string to date object, **convert to datetime first**, then to date

9.4.3 Instance Method

9.4.3.1 replace()

- Replace **year/month/day** with specified parameter, non specified params will remain unchange.

- Example below change only month. You can change year or day in combination

```
print( date.today() )
```

```
#:> 2020-12-27
```

```
print( date.today().replace(month=8) )
```

```
#:> 2020-08-27
```

9.4.3.2 weekday(), isoweekday()

For `weekday()`, Zero being Monday
For `isoweekday()`, Zero being Sunday

```
print( date.today().weekday() )
```

```
#:> 6
```

```
print( date.today().isoweekday() )
```

```
#:> 7
```

```
weekdays = ['Mon','Tue','Wed','Thu','Fri','Sat','Sun']
wd = date.today().weekday()
print( date.today(), "is day", wd ,"which is", weekdays[wd] )
```

```
#:> 2020-12-27 is day 6 which is Sun
```

9.4.3.3 Formating with isoformat()

`isoformat()` return **ISO 8601 String (YYYY-MM-DD)**

```
date.today().isoformat() # return string
```

```
#:> '2020-12-27'
```

9.4.3.4 Formating with strftime

For complete directive, see below:
https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

```
date.today().strftime("%m/%d")
```

```
#:> '12/27'
```

9.4.3.5 isocalendar()

`isocalendar` return a 3-tuple, **(ISO year, ISO week number, ISO weekday)**.

```
date.today().isocalendar() ## return tuple
```

```
#:> (2020, 52, 7)
```

9.4.4 Attributes

```
print( date.today().year )
```

```
#:> 2020
```

```
print( date.today().month )
```

```
#:> 12
```

```
print( date.today().day )
```

```
#:> 27
```

9.5 date and datetime

9.5.1 Constructor

```
import datetime as dt

print(
    dt.date(2000,1,1,), '\n',
    dt.datetime(2000,1,1,0,0,0), '\n',
    dt.datetime(year=2000,month=1,day=1,hour=23,minute=15,second=55),'\n',
    type(dt.date(2000,1,1)),'\n',
    type(dt.datetime(2000,1,1,0,0,0)))
```

```
#:> 2000-01-01
#:>  2000-01-01 00:00:00
#:>  2000-01-01 23:15:55
#:>  <class 'datetime.date'>
#:>  <class 'datetime.datetime'>
```

9.5.2 Class Method

9.5.2.1 now and today

Both `now()` and `today()` return current **system local** datetime, no timezone

```
print(  dt.datetime.now(), '\n',
        dt.datetime.now().date())
```

```
#:> 2020-12-27 16:46:19.111783
#:>  2020-12-27
```

```
dt.datetime.today()
```

```
#:> datetime.datetime(2020, 12, 27, 16, 46, 19, 123918)
```

9.5.2.2 `utcnow`

```
dt.datetime.utcnow()
```

```
#:> datetime.datetime(2020, 12, 27, 22, 46, 19, 130674)
```

9.5.2.3 `combine()` date and time

Apply `datetime.combine()` module method on both **date and time** object to get **datetime**

```
now = dt.datetime.now()
dt.datetime.combine(now.date(), now.time())
```

```
#:> datetime.datetime(2020, 12, 27, 16, 46, 19, 143439)
```

9.5.2.4 Convert from String `strptime()`

Use **`strptime`** to convert string into **datetime** object

```
%I : 12-hour
%H : 24-hour
%M : Minute
%p : AM/PM
%y : 18
%Y : 2018
%b : Mar
%m : month (1 to 12)
%d : day
```

```
datetime.strptime('2011-02-25','%Y-%m-%d')
```

```
#:> datetime.datetime(2011, 2, 25, 0, 0)
```

```
datetime.strptime('9-01-18','%d-%m-%y')
```

```
#:> datetime.datetime(2018, 1, 9, 0, 0)
```

```
datetime.strptime('09-Mar-2018','%d-%b-%Y')
```

```
#:> datetime.datetime(2018, 3, 9, 0, 0)
```

```
datetime.strptime('2/5/2018 4:49 PM', '%m/%d/%Y %I:%M %p')
```

```
#:> datetime.datetime(2018, 2, 5, 16, 49)
```

9.5.2.5 Convert from ISO `fromisoformat`

- `fromisoformat()` is intend to be reverse of `isoformat()`
- It actually **not ISO compliance**: when Z or +8 is included at the end of the string, error occur

```
#s = dt.datetime.now().isoformat()
dt.datetime.fromisoformat("2019-02-05T10:22:33")
```

```
#:> datetime.datetime(2019, 2, 5, 10, 22, 33)
```

9.5.3 Instance Method

9.5.3.1 `weekday`

```
datetime.now().weekday()
```

```
#:> 6
```

9.5.3.2 `replace`

```
datetime.now().replace(year=1999)
```

```
#:> datetime.datetime(1999, 12, 27, 16, 46, 19, 225843)
```

9.5.3.3 convert to `.time()`

```
datetime.now().time()
```

```
#:> datetime.time(16, 46, 19, 233466)
```

9.5.3.4 Convert to `.date()`

```
datetime.now().date()
```

```
#:> datetime.date(2020, 12, 27)
```

9.5.3.5 Convert to String

`str`

```
str( datetime.now() )
```

```
#:> '2020-12-27 16:46:19.256059'
```

Use `strftime()`

```
dt.datetime.now().strftime('%d-%b-%Y')
```

```
#:> '27-Dec-2020'
```

```
dt.datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%S.%fZ')  ## ISO 8601 UTC
```

```
#:> '2020-12-27T22:46:19.276268Z'
```

Use `isoformat()`

```
dt.datetime.utcnow().isoformat()
```

```
#:> '2020-12-27T22:46:19.291880'
```

9.5.4 Attributes

```
print( datetime.now().year )
```

```
#:> 2020
```

```
print( datetime.now().month )
```

```
#:> 12
```

```
print( datetime.now().day )
```

```
#:> 27
```

```
print( datetime.now().hour )
```

```
#:> 16
```

```
print( datetime.now().minute )
```

```
#:> 46
```

9.6 time

9.6.1 Constructor

```
print( time(2) )    #default single arugement, hour
```

```
#:> 02:00:00
```

```
print( time(2,15) ) #default two arguments, hour, minute
```

```
#:> 02:15:00
```

```
print( time(hour=2,minute=15,second=30) )
```

```
#:> 02:15:30
```

9.6.2 Class Method

9.6.2.1 now()

There is unfortunately no single function to extract the current time. Use **time()** function of an **datetime** object

```
datetime.now().time()
```

```
#:> datetime.time(16, 46, 19, 356190)
```

9.6.3 Attributes

```
print( datetime.now().time().hour )
```

```
#:> 16
```

```
print( datetime.now().time().minute )
```

```
#:> 46
```

```
print( datetime.now().time().second )
```

```
#:> 19
```

9.7 timedelta

- **years** argument is **not supported**
- Apply timedelta on **datetime** object
- timedelta **cannot** be applied on **time object** , because timedelta potentially go beyond single day (24H)

```
delt = timedelta(days=365,minutes=33,seconds=15)
```

```
now = datetime.now()
print ('delt+now : ', now+delt)
```

```
#:> delt+now :  2021-12-27 17:19:34.395841
```

Chapter 10

Getting External Data

Chapter 11

Plydata (dplyr for Python)

11.1 Sample Data

```
n = 200
comp = ['C' + i for i in np.random.randint( 1,4, size  = n).astype(str)] # 3x Company
dept = ['D' + i for i in np.random.randint( 1,6, size  = n).astype(str)] # 5x Department
grp =  ['G' + i for i in np.random.randint( 1,3, size  = n).astype(str)] # 2x Groups
value1 = np.random.normal( loc=50 , scale=5 , size = n)
value2 = np.random.normal( loc=20 , scale=3 , size = n)
#value3 = np.random.normal( loc=5 , scale=30 , size = n)

mydf = pd.DataFrame({
    'comp':comp,
    'dept':dept,
    'grp': grp,
    'value1':value1,
    'value2':value2
    #'value3':value3
})
mydf.head()
```

```
#:>   comp dept grp     value1     value2
#:> 0   C2   D4  G2  41.458631  18.110552
#:> 1   C2   D5  G1  50.533389  16.716333
#:> 2   C2   D3  G2  50.246733  22.670368
#:> 3   C1   D3  G2  57.352561  22.502908
#:> 4   C3   D2  G1  57.157775  20.423743
```

11.2 Column Manipulation

11.2.1 Copy Column

```
mydf >> define(newcol = 'value1')                 # simple method for one column
```

```
#:>     comp dept grp     value1     value2     newcol
#:> 0     C2   D4  G2  41.458631  18.110552  41.458631
#:> 1     C2   D5  G1  50.533389  16.716333  50.533389
#:> 2     C2   D3  G2  50.246733  22.670368  50.246733
#:> 3     C1   D3  G2  57.352561  22.502908  57.352561
#:> 4     C3   D2  G1  57.157775  20.423743  57.157775
#:> ..   ...  ...  ..        ...        ...        ...
#:> 195   C2   D5  G1  45.912737  20.610061  45.912737
#:> 196   C1   D3  G2  43.242302  26.461629  43.242302
#:> 197   C3   D4  G1  57.420667  22.227554  57.420667
#:> 198   C3   D3  G1  56.907810  23.627819  56.907810
#:> 199   C2   D5  G1  52.375667  21.806240  52.375667
#:>
#:> [200 rows x 6 columns]
```

```
mydf >> define (('newcol1', 'value1'), newcol2='value2')  # method for muiltiple new c
```

```
#:>     comp dept grp     value1     value2    newcol1    newcol2
#:> 0     C2   D4  G2  41.458631  18.110552  41.458631  18.110552
#:> 1     C2   D5  G1  50.533389  16.716333  50.533389  16.716333
#:> 2     C2   D3  G2  50.246733  22.670368  50.246733  22.670368
#:> 3     C1   D3  G2  57.352561  22.502908  57.352561  22.502908
#:> 4     C3   D2  G1  57.157775  20.423743  57.157775  20.423743
#:> ..   ...  ...  ..        ...        ...        ...        ...
#:> 195   C2   D5  G1  45.912737  20.610061  45.912737  20.610061
#:> 196   C1   D3  G2  43.242302  26.461629  43.242302  26.461629
#:> 197   C3   D4  G1  57.420667  22.227554  57.420667  22.227554
#:> 198   C3   D3  G1  56.907810  23.627819  56.907810  23.627819
#:> 199   C2   D5  G1  52.375667  21.806240  52.375667  21.806240
#:>
#:> [200 rows x 7 columns]
```

11.2.2 New Column from existing Column

Without specify the new column name, it will be derived from expression

```
mydf >> define ('value1*2')
```

```
#:>     comp dept grp     value1     value2    value1*2
#:> 0     C2   D4  G2  41.458631  18.110552   82.917261
#:> 1     C2   D5  G1  50.533389  16.716333  101.066779
```

```
#:> 2     C2   D3  G2  50.246733  22.670368  100.493466
#:> 3     C1   D3  G2  57.352561  22.502908  114.705123
#:> 4     C3   D2  G1  57.157775  20.423743  114.315550
#:> ..   ...  ...  ..        ...        ...         ...
#:> 195   C2   D5  G1  45.912737  20.610061   91.825473
#:> 196   C1   D3  G2  43.242302  26.461629   86.484604
#:> 197   C3   D4  G1  57.420667  22.227554  114.841335
#:> 198   C3   D3  G1  56.907810  23.627819  113.815620
#:> 199   C2   D5  G1  52.375667  21.806240  104.751334
#:>
#:> [200 rows x 6 columns]
```

Specify the new column name

```
mydf >> define(value3 = 'value1*2')
```

```
#:>     comp dept grp     value1     value2      value3
#:> 0     C2   D4  G2  41.458631  18.110552   82.917261
#:> 1     C2   D5  G1  50.533389  16.716333  101.066779
#:> 2     C2   D3  G2  50.246733  22.670368  100.493466
#:> 3     C1   D3  G2  57.352561  22.502908  114.705123
#:> 4     C3   D2  G1  57.157775  20.423743  114.315550
#:> ..   ...  ...  ..        ...        ...         ...
#:> 195   C2   D5  G1  45.912737  20.610061   91.825473
#:> 196   C1   D3  G2  43.242302  26.461629   86.484604
#:> 197   C3   D4  G1  57.420667  22.227554  114.841335
#:> 198   C3   D3  G1  56.907810  23.627819  113.815620
#:> 199   C2   D5  G1  52.375667  21.806240  104.751334
#:>
#:> [200 rows x 6 columns]
```

Define **multiple** new columns in one go. Observe there are three ways to specify the new columns

```
mydf >> define('value1*2',('newcol2','value2*2'),newcol3='value2*3')
```

```
#:>     comp dept grp     value1     value2    value1*2    newcol2    newcol3
#:> 0     C2   D4  G2  41.458631  18.110552   82.917261  36.221105  54.331657
#:> 1     C2   D5  G1  50.533389  16.716333  101.066779  33.432667  50.149000
#:> 2     C2   D3  G2  50.246733  22.670368  100.493466  45.340735  68.011103
#:> 3     C1   D3  G2  57.352561  22.502908  114.705123  45.005816  67.508725
#:> 4     C3   D2  G1  57.157775  20.423743  114.315550  40.847486  61.271229
#:> ..   ...  ...  ..        ...        ...         ...        ...        ...
#:> 195   C2   D5  G1  45.912737  20.610061   91.825473  41.220122  61.830182
#:> 196   C1   D3  G2  43.242302  26.461629   86.484604  52.923257  79.384886
#:> 197   C3   D4  G1  57.420667  22.227554  114.841335  44.455108  66.682663
#:> 198   C3   D3  G1  56.907810  23.627819  113.815620  47.255638  70.883456
#:> 199   C2   D5  G1  52.375667  21.806240  104.751334  43.612479  65.418719
```

```
#:>
#:> [200 rows x 8 columns]
```

11.2.3 Select Column(s)

```
mydf2 = mydf >> define(newcol1='value1',newcol2='value2')
mydf2.info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 200 entries, 0 to 199
#:> Data columns (total 7 columns):
#:>  #   Column   Non-Null Count  Dtype
#:> ---  ------   --------------  -----
#:>  0   comp     200 non-null    object
#:>  1   dept     200 non-null    object
#:>  2   grp      200 non-null    object
#:>  3   value1   200 non-null    float64
#:>  4   value2   200 non-null    float64
#:>  5   newcol1  200 non-null    float64
#:>  6   newcol2  200 non-null    float64
#:> dtypes: float64(4), object(3)
#:> memory usage: 11.1+ KB
```

11.2.3.1 By Column Names

Exact Coumn Name

```
mydf2 >> select ('comp','dept','value1')
```

```
#:>     comp dept     value1
#:> 0     C2   D4  41.458631
#:> 1     C2   D5  50.533389
#:> 2     C2   D3  50.246733
#:> 3     C1   D3  57.352561
#:> 4     C3   D2  57.157775
#:> ..   ...  ...        ...
#:> 195   C2   D5  45.912737
#:> 196   C1   D3  43.242302
#:> 197   C3   D4  57.420667
#:> 198   C3   D3  56.907810
#:> 199   C2   D5  52.375667
#:>
#:> [200 rows x 3 columns]
```

Column Name Starts With ...

```
mydf2 >> select ('comp', startswith='val')
```

```
#:>     comp     value1     value2
#:> 0     C2  41.458631  18.110552
#:> 1     C2  50.533389  16.716333
#:> 2     C2  50.246733  22.670368
#:> 3     C1  57.352561  22.502908
#:> 4     C3  57.157775  20.423743
#:> ..   ...        ...        ...
#:> 195   C2  45.912737  20.610061
#:> 196   C1  43.242302  26.461629
#:> 197   C3  57.420667  22.227554
#:> 198   C3  56.907810  23.627819
#:> 199   C2  52.375667  21.806240
#:>
#:> [200 rows x 3 columns]
```

Column Name Ends With …

```
mydf2 >> select ('comp',endswith=('1','2','3'))
```

```
#:>     comp     value1     value2    newcol1    newcol2
#:> 0     C2  41.458631  18.110552  41.458631  18.110552
#:> 1     C2  50.533389  16.716333  50.533389  16.716333
#:> 2     C2  50.246733  22.670368  50.246733  22.670368
#:> 3     C1  57.352561  22.502908  57.352561  22.502908
#:> 4     C3  57.157775  20.423743  57.157775  20.423743
#:> ..   ...        ...        ...        ...        ...
#:> 195   C2  45.912737  20.610061  45.912737  20.610061
#:> 196   C1  43.242302  26.461629  43.242302  26.461629
#:> 197   C3  57.420667  22.227554  57.420667  22.227554
#:> 198   C3  56.907810  23.627819  56.907810  23.627819
#:> 199   C2  52.375667  21.806240  52.375667  21.806240
#:>
#:> [200 rows x 5 columns]
```

Column Name Contains …

```
mydf2 >> select('comp', contains=('col','val'))
```

```
#:>     comp     value1     value2    newcol1    newcol2
#:> 0     C2  41.458631  18.110552  41.458631  18.110552
#:> 1     C2  50.533389  16.716333  50.533389  16.716333
#:> 2     C2  50.246733  22.670368  50.246733  22.670368
#:> 3     C1  57.352561  22.502908  57.352561  22.502908
#:> 4     C3  57.157775  20.423743  57.157775  20.423743
#:> ..   ...        ...        ...        ...        ...
#:> 195   C2  45.912737  20.610061  45.912737  20.610061
```

```
#:> 196   C1  43.242302  26.461629  43.242302  26.461629
#:> 197   C3  57.420667  22.227554  57.420667  22.227554
#:> 198   C3  56.907810  23.627819  56.907810  23.627819
#:> 199   C2  52.375667  21.806240  52.375667  21.806240
#:>
#:> [200 rows x 5 columns]
```

11.2.3.2 Specify Column Range

```
mydf2 >> select ('comp', slice('value1','newcol2'))
```

```
#:>     comp     value1     value2    newcol1    newcol2
#:> 0     C2  41.458631  18.110552  41.458631  18.110552
#:> 1     C2  50.533389  16.716333  50.533389  16.716333
#:> 2     C2  50.246733  22.670368  50.246733  22.670368
#:> 3     C1  57.352561  22.502908  57.352561  22.502908
#:> 4     C3  57.157775  20.423743  57.157775  20.423743
#:> ..   ...        ...        ...        ...        ...
#:> 195   C2  45.912737  20.610061  45.912737  20.610061
#:> 196   C1  43.242302  26.461629  43.242302  26.461629
#:> 197   C3  57.420667  22.227554  57.420667  22.227554
#:> 198   C3  56.907810  23.627819  56.907810  23.627819
#:> 199   C2  52.375667  21.806240  52.375667  21.806240
#:>
#:> [200 rows x 5 columns]
```

11.2.4 Drop Column(s)

```
mydf2 >> select('newcol1','newcol2',drop=True)
```

```
#:>     comp dept grp     value1     value2
#:> 0     C2   D4  G2  41.458631  18.110552
#:> 1     C2   D5  G1  50.533389  16.716333
#:> 2     C2   D3  G2  50.246733  22.670368
#:> 3     C1   D3  G2  57.352561  22.502908
#:> 4     C3   D2  G1  57.157775  20.423743
#:> ..   ...  ...  ..        ...        ...
#:> 195   C2   D5  G1  45.912737  20.610061
#:> 196   C1   D3  G2  43.242302  26.461629
#:> 197   C3   D4  G1  57.420667  22.227554
#:> 198   C3   D3  G1  56.907810  23.627819
#:> 199   C2   D5  G1  52.375667  21.806240
#:>
#:> [200 rows x 5 columns]
```

```
mydf >> rename( {'val.1' : 'value1',
                 'val.2' : 'value2' })
```

```
#:>     comp dept grp      val.1      val.2
#:> 0     C2   D4  G2  41.458631  18.110552
#:> 1     C2   D5  G1  50.533389  16.716333
#:> 2     C2   D3  G2  50.246733  22.670368
#:> 3     C1   D3  G2  57.352561  22.502908
#:> 4     C3   D2  G1  57.157775  20.423743
#:> ..   ...  ...  ..        ...        ...
#:> 195   C2   D5  G1  45.912737  20.610061
#:> 196   C1   D3  G2  43.242302  26.461629
#:> 197   C3   D4  G1  57.420667  22.227554
#:> 198   C3   D3  G1  56.907810  23.627819
#:> 199   C2   D5  G1  52.375667  21.806240
#:>
#:> [200 rows x 5 columns]
```

Combined Method
Combine both assignment and dictionary method

```
mydf >> rename( {'val.1' : 'value1',
                 'val.2' : 'value2'
              }, group = 'grp' )
```

```
#:>     comp dept group      val.1      val.2
#:> 0     C2   D4    G2  41.458631  18.110552
#:> 1     C2   D5    G1  50.533389  16.716333
#:> 2     C2   D3    G2  50.246733  22.670368
#:> 3     C1   D3    G2  57.352561  22.502908
#:> 4     C3   D2    G1  57.157775  20.423743
#:> ..   ...  ...   ...        ...        ...
#:> 195   C2   D5    G1  45.912737  20.610061
#:> 196   C1   D3    G2  43.242302  26.461629
#:> 197   C3   D4    G1  57.420667  22.227554
#:> 198   C3   D3    G1  56.907810  23.627819
#:> 199   C2   D5    G1  52.375667  21.806240
#:>
#:> [200 rows x 5 columns]
```

11.3 Sorting (arrange)

Use **'-colName'** for decending

```
mydf >> arrange('comp', '-value1')
```

```
#:>     comp dept grp     value1     value2
```

```
#:> 95    C1   D3  G2  61.362872  15.803437
#:> 5     C1   D4  G2  61.352595  23.016850
#:> 169   C1   D1  G1  58.803351  20.668507
#:> 30    C1   D2  G2  58.482879  19.045717
#:> 177   C1   D1  G2  57.726445  20.277626
#:> ..   ...  ...  ..        ...        ...
#:> 191   C3   D5  G2  43.204717  25.155417
#:> 41    C3   D1  G2  41.919407  16.717460
#:> 68    C3   D5  G2  41.356681  26.080992
#:> 129   C3   D4  G1  40.663750  26.141246
#:> 168   C3   D5  G2  37.414075  21.820749
#:>
#:> [200 rows x 5 columns]
```

11.4 Grouping

```
mydf.info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 200 entries, 0 to 199
#:> Data columns (total 5 columns):
#:>  #   Column  Non-Null Count  Dtype
#:> ---  ------  --------------  -----
#:>  0   comp    200 non-null    object
#:>  1   dept    200 non-null    object
#:>  2   grp     200 non-null    object
#:>  3   value1  200 non-null    float64
#:>  4   value2  200 non-null    float64
#:> dtypes: float64(2), object(3)
#:> memory usage: 7.9+ KB
```

```
gdf = mydf >> group_by('comp','dept')
type(gdf)
```

```
#:> <class 'plydata.types.GroupedDataFrame'>
```

11.5 Summarization

11.5.1 Simple Method

Passing Multiple Expressions

```
gdf >> summarize('n()','sum(value1)','mean(value2)')
```

11.5.2 Specify Summarized Column Name

Assignment Method
- Passing colName='expression'**
- Column name cannot contain special character

```
gdf >> summarize(count='n()',v1sum='sum(value1)',v2_mean='mean(value2)')
```

Tuple Method ('colName','expression')
Use when the column name contain special character

```
gdf >> summarize(('count','n()'),('v1.sum','sum(value1)'),('s2.sum','sum(value2)'),v2mean=np.mean
```

11.5.3 Number of Rows in Group

- n() : total rows in group

- n_unique() : total of rows with unique value

```
gdf >> summarize(count='n()', va11_unique='n_unique(value1)')
```

Chapter 12

numpy

- Best array data manipulation, fast
- numpy array allows only single data type, unlike list
- Support matrix operation

12.1 Environment Setup

```
import pandas as pd
import matplotlib.pyplot as plt
import math
pd.set_option( 'display.notebook_repr_html', False)  # render Series and DataFrame as text, not H
pd.set_option( 'display.max_column', 10)    # number of columns
pd.set_option( 'display.max_rows', 10)     # number of rows
pd.set_option( 'display.width', 90)        # number of characters per row
```

12.2 Module Import

```
import numpy as np
np.__version__

## other modules
```

```
#:> '1.19.1'
```

```
from datetime import datetime
from datetime import date
from datetime import time
```

12.3 Data Types

12.3.1 NumPy Data Types

NumPy supports a much greater variety of numerical types than Python does. This makes numpy **much more powerful** https://www.numpy.org/devdocs/user/basics.types.html

Integer: np.int8, np.int16, np.int32, np.uint8, np.uint16, np.uint32
Float: np.float32, np.float64

12.3.2 int32/64

`np.int` is actually **python standard int**

```
x = np.int(13)
y = int(13)
print( type(x) )
```

```
#:> <class 'int'>
```

```
print( type(y) )
```

```
#:> <class 'int'>
```

`np.int32/64` are NumPy specific

```
x = np.int32(13)
y = np.int64(13)
print( type(x) )
```

```
#:> <class 'numpy.int32'>
```

```
print( type(y) )
```

```
#:> <class 'numpy.int64'>
```

12.3.3 float32/64

```
x = np.float(13)
y = float(13)
print( type(x) )
```

```
#:> <class 'float'>
```

```
print( type(y) )
```

```
#:> <class 'float'>
```

```
x = np.float32(13)
y = np.float64(13)
print( type(x) )
```

```
#:> <class 'numpy.float32'>
```

```
print( type(y) )
```

```
#:> <class 'numpy.float64'>
```

12.3.4 bool

`np.bool` is actually **python standard bool**

```
x = np.bool(True)
print( type(x) )
```

```
#:> <class 'bool'>
```

```
print( type(True) )
```

```
#:> <class 'bool'>
```

12.3.5 str

`np.str` is actually **python standard str**

```
x = np.str("ali")
print( type(x) )
```

```
#:> <class 'str'>
```

```
x = np.str_("ali")
print( type(x) )
```

```
#:> <class 'numpy.str_'>
```

12.3.6 datetime64

Unlike python standard datetime library, there is **no seperation** of date, datetime and time.
There is **no time equivalent object**
NumPy only has one object: **datetime64** object .

12.3.6.1 Constructor

From String
Note that the input string **cannot be ISO8601 compliance**, meaning any timezone related information at the end of the string (such as Z or +8) will result in **error**.

```
np.datetime64('2005-02')
```

```
#:> numpy.datetime64('2005-02')
```

```
np.datetime64('2005-02-25')
```

```
#:> numpy.datetime64('2005-02-25')
```

```
np.datetime64('2005-02-25T03:30')
```

```
#:> numpy.datetime64('2005-02-25T03:30')
```

From datetime

```
np.datetime64( date.today() )
```

```
#:> numpy.datetime64('2020-12-27')
```

```
np.datetime64( datetime.now() )
```

```
#:> numpy.datetime64('2020-12-27T16:46:22.485797')
```

12.3.6.2 Instance Method

Convert to **datetime** using **`astype()`**

```
dt64 = np.datetime64("2019-01-31" )
dt64.astype(datetime)
```

```
#:> datetime.date(2019, 1, 31)
```

12.3.7 nan

12.3.7.1 Creating NaN

NaN is NOT A BUILT-IN datatype. It means **not a number**, a numpy **float** object type. Can be created using two methods below.

```
import numpy as np
import pandas as pd
import math

kosong1 = float('NaN')
kosong2 = np.nan

print('Type: ', type(kosong1), '\n',
       'Value: ', kosong1)
```

```
#:> Type:  <class 'float'>
#:>  Value:  nan
```

```
print('Type: ', type(kosong2), '\n',
       'Value: ', kosong2)
```

```
#:> Type:  <class 'float'>
#:>  Value:  nan
```

12.3.7.2 Detecting NaN

Detect nan using various function from panda, numpy and math.

```
print(pd.isna(kosong1), '\n',
      pd.isna(kosong2), '\n',
      np.isnan(kosong1),'\n',
      math.isnan(kosong2))
```

```
#:> True
#:>  True
#:>  True
#:>  True
```

12.3.7.3 Operation

```

print( True and kosong1,
       kosong1 and True)
```

12.3.7.3.1 Logical Operator

```
#:> nan True
```

```
print( True or kosong1,
       False or kosong1)
```

```
#:> True nan
```

12.3.7.3.2 Comparing Compare nan with **anything results in False**, including itself.

```
print(kosong1 > 0, kosong1==0, kosong1<0,
      kosong1 ==1, kosong1==kosong1, kosong1==False, kosong1==True)
```

```
#:> False False False False False False False
```

12.3.7.3.3 Casting nan is numpy floating value. It is not a zero value, therefore casting to boolean returns True.

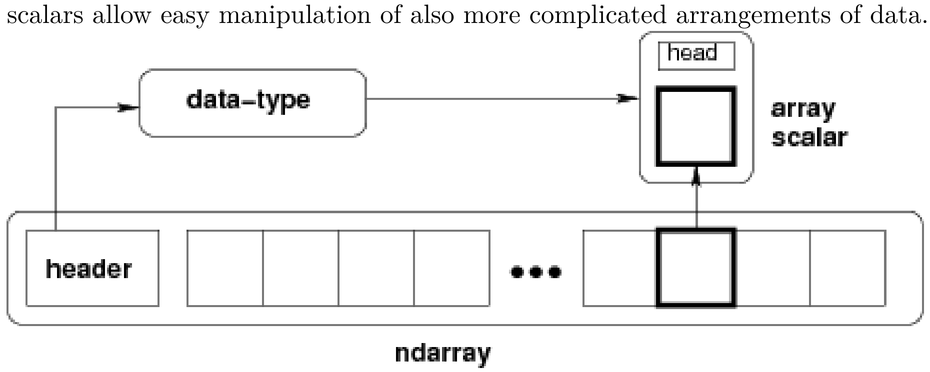
```
bool(kosong1)
```

```
#:> True
```

12.4 Numpy Array

12.4.1 Concept

Structure - NumPy provides an N-dimensional array type, the **ndarray** - **ndarray** is **homogenous**: every item takes up the same size block of memory, and all blocks - For each ndarray, there is a seperate **dtype object**, which describe ndarray data type
- An item extracted from an array, e.g., by indexing, is represented by a Python object whose type is one of the array scalar types built in NumPy. The array scalars allow easy manipulation of also more complicated arrangements of data.



12.4.2 Constructor

By default, numpy.array autodetect its data types based on most common denominator

12.4.2.1 dType: int, float

Notice example below **auto detected** as int32 data type

```
x = np.array( (1,2,3,4,5) )
print(x)
```

```
#:> [1 2 3 4 5]
```

```
print('Type: ', type(x))
```

```
#:> Type:  <class 'numpy.ndarray'>
```

```
print('dType:', x.dtype)
```

```
#:> dType: int64
```

Notice example below **auto detected** as float64 data type

```
x = np.array( (1,2,3,4.5,5) )
print(x)
```

```
# print('Type: ', type(x))
# print('dType:', x.dtype)
```

```
#:> [1.  2.  3.  4.5 5. ]
```

You can specify dtype to specify desired data types.
NumPy will **auto convert** the data into specifeid types. Observe below that we convert float into integer

```
x = np.array( (1,2,3,4.5,5), dtype='int' )
print(x)
```

```
#:> [1 2 3 4 5]
```

```
print('Type: ', type(x))
```

```
#:> Type:  <class 'numpy.ndarray'>
```

```
print('dType:', x.dtype)
```

```
#:> dType: int64
```

12.4.2.2 dType: datetime64

Specify `dtype` is necessary to ensure output is datetime type. If not, output is generic **object** type.

From `str`

```
x = np.array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64')
print(x)
```

```
#:> ['2007-07-13' '2006-01-13' '2010-08-13']
```

```
print('Type: ', type(x))
```

```
#:> Type:  <class 'numpy.ndarray'>
```

```
print('dType:', x.dtype)
```

```
#:> dType: datetime64[D]
```

From `datetime`

```
x = np.array([datetime(2019,1,12), datetime(2019,1,14),datetime(2019,3,3)], dtype='datetime64')
print(x)
```

```
#:> ['2019-01-12T00:00:00.000000' '2019-01-14T00:00:00.000000'
#:>  '2019-03-03T00:00:00.000000']
```

```
print('Type: ', type(x))
```

```
#:> Type:  <class 'numpy.ndarray'>
```

```
print('dType:', x.dtype)
```

```
#:> dType: datetime64[us]
```

```
print('\nElement Type:',type(x[1]))
```

```
#:>
#:> Element Type: <class 'numpy.datetime64'>
```

12.4.2.3 2D Array

```
x = np.array([range(10),np.arange(10)])
x
```

```
#:> array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
#:>        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

12.4.3 Dimensions

12.4.3.1 Differentiating Dimensions

1-D array is array of single list
2-D array is array made of list containing lists (each row is a list)
2-D single row array is array with list containing just one list

12.4.3.2 1-D Array

Observe that the **shape of the array** is (5,). It seems like an array with 5 rows, **empty columns** !
What it really means is 5 items **single dimension**.

```
arr = np.array(range(5))
print (arr)
```

```
#:> [0 1 2 3 4]
```

```
print (arr.shape)
```

```
#:> (5,)
```

```
print (arr.ndim)
```

```
#:> 1
```

12.4.3.3 2-D Array

```
arr = np.array([range(5),range(5,10),range(10,15)])
print (arr)
```

```
#:> [[ 0  1  2  3  4]
#:>  [ 5  6  7  8  9]
#:>  [10 11 12 13 14]]
```

```
print (arr.shape)
```

```
#:> (3, 5)
```

```
print (arr.ndim)
```

```
#:> 2
```

12.4.3.4 2-D Array - Single Row

```
arr = np.array([range(5)])
print (arr)
```

```
#:> [[0 1 2 3 4]]
```

```
print (arr.shape)
```

```
#:> (1, 5)
```

```
print (arr.ndim)
```

```
#:> 2
```

12.4.3.5 2-D Array : Single Column

Using array slicing method with **newaxis** at **COLUMN**, will turn 1D array into 2D of **single column**

```
arr = np.arange(5)[:, np.newaxis]
print (arr)
```

```
#:> [[0]
#:>  [1]
#:>  [2]
#:>  [3]
#:>  [4]]
```

```
print (arr.shape)
```

```
#:> (5, 1)
```

```
print (arr.ndim)
```

```
#:> 2
```

Using array slicing method with **newaxis** at **ROW**, will turn 1D array into 2D of **single row**

```
arr = np.arange(5)[np.newaxis,:]
print (arr)
```

```
#:> [[0 1 2 3 4]]
```

```
print (arr.shape)
```

```
#:> (1, 5)
```

```
print (arr.ndim)
```

```
#:> 2
```

12.4.4 Class Method

12.4.4.1 arange()

Generate array with a sequence of numbers

```
np.arange(10)
```

```
#:> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

12.4.4.2 ones()

```
np.ones(10)  # One dimension, default is float
```

```
#:> array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
np.ones((2,5),'int')  #Two dimensions
```

```
#:> array([[1, 1, 1, 1, 1],
#:>        [1, 1, 1, 1, 1]])
```

12.4.4.3 zeros()

```
np.zeros( 10 )    # One dimension, default is float
```

```
#:> array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
np.zeros((2,5),'int')   # 2 rows, 5 columns of ZERO
```

```
#:> array([[0, 0, 0, 0, 0],
#:>        [0, 0, 0, 0, 0]])
```

12.4.4.4 where()

On **1D array** `numpy.where()` returns the items matching the criteria

```
ar1 = np.array(range(10))
print( ar1 )
```

```
#:> [0 1 2 3 4 5 6 7 8 9]
```

```
print( np.where(ar1>5) )
```

```
#:> (array([6, 7, 8, 9]),)
```

On **2D array**, `where()` return array of **row index and col index** for matching elements

```
ar = np.array([(1,2,3,4,5),(11,12,13,14,15),(21,22,23,24,25)])
print ('Data : \n', ar)
```

```
#:> Data :
#:>  [[ 1  2  3  4  5]
#:>  [11 12 13 14 15]
#:>  [21 22 23 24 25]]
```

```
np.where(ar>13)
```

```
#:> (array([1, 1, 2, 2, 2, 2, 2]), array([3, 4, 0, 1, 2, 3, 4]))
```

12.4.4.5 Logical Methods

`numpy.logical_or`
Perform **or** operation on two boolean array, generate new resulting **boolean arrays**

```
ar = np.arange(10)
print( ar==3 )  # boolean array 1
```

```
#:> [False False False  True False False False False False False]
```

```
print( ar==6 )  # boolean array 2
```

```
#:> [False False False False False False  True False False False]
```

```
print( np.logical_or(ar==3,ar==6 ) ) # resulting boolean
```

```
#:> [False False False  True False False  True False False False]
```

`numpy.logical_and`
Perform **and** operation on two boolean array, generate new resulting **boolean arrays**

```
ar = np.arange(10)
print( ar==3 ) # boolean array 1
```

```
#:> [False False False  True False False False False False False]
```

```
print( ar==6 ) # boolean array 2
```

```
#:> [False False False False False False  True False False False]
```

```
print( np.logical_and(ar==3,ar==6 ) )  # resulting boolean
```

```
#:> [False False False False False False False False False False]
```

12.4.5 Instance Method

12.4.5.1 astype() conversion

Convert to from datetime64 to datetime

```
ar1 = np.array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64')
print( type(ar1) )  ## a numpy array
```

```
#:> <class 'numpy.ndarray'>
```

```
print( ar1.dtype )  ## dtype is a numpy data type
```

```
#:> datetime64[D]
```

After convert to datetime (non-numpy object, the dtype becomes **generic 'object'**.

```
ar2 = ar1.astype(datetime)
print( type(ar2) )  ## still a numpy array
```

```
#:> <class 'numpy.ndarray'>
```

```
print( ar2.dtype )  ## dtype becomes generic 'object'
```

```
#:> object
```

12.4.5.2 reshape()

`reshape ( row numbers, col numbers )`

Sample Data

```
a = np.array([range(5), range(10,15), range(20,25), range(30,35)])
a
```

```
#:> array([[ 0,  1,  2,  3,  4],
#:>        [10, 11, 12, 13, 14],
#:>        [20, 21, 22, 23, 24],
#:>        [30, 31, 32, 33, 34]])
```

Resphepe 1-Dim to 2-Dim

```
np.arange(12) # 1-D Array
```

```
#:> array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
np.arange(12).reshape(3,4)  # 2-D Array
```

```
#:> array([[ 0,  1,  2,  3],
#:>        [ 4,  5,  6,  7],
#:>        [ 8,  9, 10, 11]])
```

Respahe 2-Dim to 2-Dim

```
np.array([range(5), range(10,15)])  # 2-D Array
```

```
#:> array([[ 0,  1,  2,  3,  4],
#:>        [10, 11, 12, 13, 14]])
```

```
np.array([range(5), range(10,15)]).reshape(5,2) # 2-D Array
```

```
#:> array([[ 0,  1],
#:>        [ 2,  3],
#:>        [ 4, 10],
#:>        [11, 12],
#:>        [13, 14]])
```

Reshape 2-Dimension to 2-Dim (of single row) - Change 2x10 to 1x10 - Observe [[ ]], and the number of dimension is stll 2, don't be fooled

```
np.array( [range(0,5), range(5,10)])  # 2-D Array
```

```
#:> array([[0, 1, 2, 3, 4],
#:>        [5, 6, 7, 8, 9]])
```

```
np.array( [range(0,5), range(5,10)]).reshape(1,10) # 2-D Array
```

```
#:> array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

Reshape 1-Dim Array to 2-Dim Array (single column)

```
np.arange(8)
```

```
#:> array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
np.arange(8).reshape(8,1)
```

```
#:> array([[0],
#:>        [1],
#:>        [2],
#:>        [3],
#:>        [4],
#:>        [5],
#:>        [6],
#:>        [7]])
```

A better method, use **newaxis**, easier because no need to input row number as parameter

```
np.arange(8)[:,np.newaxis]
```

```
#:> array([[0],
#:>        [1],
#:>        [2],
#:>        [3],
#:>        [4],
#:>        [5],
#:>        [6],
#:>        [7]])
```

Reshape 1-Dim Array to 2-Dim Array (single row)

```
np.arange(8)
```

```
#:> array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
np.arange(8)[np.newaxis,:]
```

```
#:> array([[0, 1, 2, 3, 4, 5, 6, 7]])
```

12.4.6 Element Selection

12.4.6.1 Sample Data

```
x1 = np.array( (0,1,2,3,4,5,6,7,8))
x2 = np.array(( (1,2,3,4,5),
      (11,12,13,14,15),
      (21,22,23,24,25)))
print(x1)
```

```
#:> [0 1 2 3 4 5 6 7 8]
```

```
print(x2)
```

```
#:> [[ 1  2  3  4  5]
#:>  [11 12 13 14 15]
#:>  [21 22 23 24 25]]
```

12.4.6.2 1-Dimension

All indexing starts from 0 (not 1)

Choosing **Single Element** does not return array

```
print( x1[0]   )  ## first element
```

```
#:> 0
```

```
print( x1[-1]  )  ## last element
```

```
#:> 8
```

```
print( x1[3]   )  ## third element from start 3
```

```
#:> 3
```

```
print( x1[-3]  )  ## third element from end
```

```
#:> 6
```

Selecting **multiple elments** return **ndarray**

```
print( x1[:3]  )  ## first 3 elements
```

```
#:> [0 1 2]
```

```
print( x1[-3:])   ## last 3 elements
```

```
#:> [6 7 8]
```

```
print( x1[3:]  )  ## all except first 3 elements
```

```
#:> [3 4 5 6 7 8]
```

```
print( x1[:-3] )  ## all except last 3 elements
```

```
#:> [0 1 2 3 4 5]
```

```
print( x1[1:4] )  ## elemnt 1 to 4 (not including 4)
```

```
#:> [1 2 3]
```

12.4.6.3 2-Dimension

Indexing with [**row_positoins, row_positions**], index starts with 0

```
x[1:3, 1:4] # row 1 to 2 column 1 to 3
```

```
#:> array([[1, 2, 3]])
```

12.4.7 Attributes

12.4.7.1 `dtype`

ndarray contain a property called **dtype**, whcih tell us the type of underlying items

```
a = np.array( (1,2,3,4,5), dtype='float' )
a.dtype
```

```
#:> dtype('float64')
```

```
print(a.dtype)
```

```
#:> float64
```

```
print( type(a[1]))
```

```
#:> <class 'numpy.float64'>
```

12.4.7.2 `dim`

`dim` returns the number of dimensions of the NumPy array. Example below shows 2-D array

```
x = np.array(( (1,2,3,4,5),
      (11,12,13,14,15),
      (21,22,23,24,25)))
x.ndim
```

```
#:> 2
```

12.4.7.3 `shape`

`shape` return a type of **(rows, cols)**

```
x = np.array(( (1,2,3,4,5),
      (11,12,13,14,15),
      (21,22,23,24,25)))
x.shape
```

```
#:> (3, 5)
```

```
np.identity(5)
```

```
#:> array([[1., 0., 0., 0., 0.],
#:>        [0., 1., 0., 0., 0.],
#:>        [0., 0., 1., 0., 0.],
#:>        [0., 0., 0., 1., 0.],
#:>        [0., 0., 0., 0., 1.]])
```

12.4.8 Operations

12.4.8.1 Arithmetic

Sample Date

```
ar = np.arange(10)
print( ar )
```

```
#:> [0 1 2 3 4 5 6 7 8 9]
```

*****

```
ar = np.arange(10)
print (ar)
```

```
#:> [0 1 2 3 4 5 6 7 8 9]
```

```
print (ar*2)
```

```
#:> [ 0  2  4  6  8 10 12 14 16 18]
```

+ and -*

```
ar = np.arange(10)
print (ar+2)
```

```
#:> [ 2  3  4  5  6  7  8  9 10 11]
```

```
print (ar-2)
```

```
#:> [-2 -1  0  1  2  3  4  5  6  7]
```

12.4.8.2 Comparison

Sample Data

```
ar = np.arange(10)
print( ar )
```

```
#:> [0 1 2 3 4 5 6 7 8 9]
```

==

```
print( ar==3 )
```

```
#:> [False False False  True False False False False False False]
```

>, >=, <, <=

```
print( ar>3 )
```

```
#:> [False False False False  True  True  True  True  True  True]
```

```
print( ar<=3 )
```

```
#:> [ True  True  True  True False False False False False False]
```

12.5 Random Numbers

12.5.1 Uniform Distribution

12.5.1.1 Random Integer (with Replacement)

randint() Return random integers from **low (inclusive) to high (exclusive)**

```
np.random.randint( low )                  # generate an integer, i, which         i < l
np.random.randint( low, high )            # generate an integer, i, which  low <= i < h
np.random.randint( low, high, size=1)     # generate an ndarray of integer, single dime
np.random.randint( low, high, size=(r,c)) # generate an ndarray of integer, two dimensi
```

```
np.random.randint( 10 )
```

```
#:> 5
```

```
np.random.randint( 10, 20 )
```

```
#:> 11
```

```
np.random.randint( 10, high=20, size=5)   # single dimension
```

```
#:> array([11, 10, 10, 10, 13])
```

```
np.random.randint( 10, 20, (3,5) )        # two dimensions
```

```
#:> array([[14, 18, 10, 13, 19],
#:>        [14, 15, 12, 11, 10],
#:>        [17, 14, 19, 12, 18]])
```

12.5.1.2 Random Integer (with or without replacement)

```
numpy.random .choice( a, size, replace=True)
 # sampling from a,
 #   if a is integer, then it is assumed sampling from arange(a)
 #   if a is an 1-D array, then sampling from this array
```

```
np.random.choice(10,5, replace=False) # take 5 samples from 0:19, without replacement
```

```
#:> array([6, 5, 4, 9, 8])
```

```
np.random.choice( np.arange(10,20), 5, replace=False)
```

```
#:> array([14, 16, 10, 12, 19])
```

12.5.1.3 Random Float

randf() Generate float numbers in **between 0.0 and 1.0**

```
np.random.ranf(size=None)
```

```
np.random.ranf(4)
```

```
#:> array([0.49160403, 0.49473349, 0.83100596, 0.5301932 ])
```

uniform() Return random float from **low (inclusive) to high (exclusive)**

```
np.random.uniform( low )                  # generate an float, i, which         f < low
np.random.uniform( low, high )            # generate an float, i, which  low <= f < hig
np.random.uniform( low, high, size=1)     # generate an array of float, single dimensio
```

```
np.random.uniform( low, high, size=(r,c)) # generate an array of float, two dimensions
```

```
np.random.uniform( 2 )
```

```
#:> 1.2185430413647698
```

```
np.random.uniform( 2,5, size=(4,4) )
```

```
#:> array([[3.2690297 , 3.19028522, 2.17089908, 4.68600169],
#:>        [3.1016267 , 3.33146145, 4.85362066, 2.33014468],
#:>        [3.34691159, 2.56173585, 4.07680839, 3.28146572],
#:>        [4.10441737, 4.32767271, 2.45009958, 4.45380907]])
```

12.5.2 Normal Distribution

```
numpy. random.randn (n_items)       # 1-D standard normal (mean=0, stdev=1)
numpy. random.randn (nrows, ncols)  # 2-D standard normal (mean=0, stdev=1)
numpy. random.standard_normal( size=None )                # default to mean = 0, stdev = 1, non-
configurable
numpy. random.normal         ( loc=0, scale=1, size=None) # loc = mean, scale = stdev, size = dim
```

12.5.2.1 Standard Normal Distribution

Generate random normal numbers with gaussion distribution (mean=0, stdev=1)

One Dimension

```
np.random.standard_normal(3)
```

```
#:> array([0.09046236, 0.26511925, 0.35930158])
```

```
np.random.randn(3)
```

```
#:> array([-1.95232954, -2.0769775 ,  0.93544838])
```

Two Dimensions

```
np.random.randn(2,4)
```

```
#:> array([[-0.27228162, -0.47069835, -1.11330727, -0.47857133],
#:>        [ 1.16462632, -1.53560825, -0.05827337, -1.20269362]])
```

```
np.random.standard_normal((2,4))
```

```
#:> array([[-0.75010435, -1.51424052,  1.33702183, -0.41098278],
#:>        [-0.63992714, -0.52381271, -1.16224483,  0.17600686]])
```

Observe: randn(), standard_normal() and normal() are able to generate standard normal numbers

```
np.random.seed(15)
print (np.random.randn(5))
```

```
#:> [-0.31232848  0.33928471 -0.15590853 -0.50178967  0.23556889]
```

```
np.random.seed(15)
print (np.random.normal ( size = 5 )) # stdev and mean not specified, default to stand
```

```
#:> [-0.31232848  0.33928471 -0.15590853 -0.50178967  0.23556889]
```

```
np.random.seed(15)
print (np.random.standard_normal (size=5))
```

```
#:> [-0.31232848  0.33928471 -0.15590853 -0.50178967  0.23556889]
```

12.5.2.2 Normal Distribution (Non-Standard)

```
np.random.seed(125)
np.random.normal( loc = 12, scale=1.25, size=(3,3))
```

```
#:> array([[11.12645382, 12.01327885, 10.81651695],
#:>        [12.41091248, 12.39383072, 11.49647195],
#:>        [ 8.70837035, 12.25246312, 11.49084235]])
```

12.5.2.3 Linear Spacing

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
# endpoint: If True, stop is the last sample, otherwise it is not included
```

Include Endpoint
Step = Gap divide by (number of elements minus 1) (2/(10-1))

```
np.linspace(1,3,10) #default endpont=True
```

```
#:> array([1.        , 1.22222222, 1.44444444, 1.66666667, 1.88888889,
#:>        2.11111111, 2.33333333, 2.55555556, 2.77777778, 3.        ])
```

Does Not Include Endpoint
Step = Gap divide by (number of elements minus 1) (2/(101))

```
np.linspace(1,3,10,endpoint=False)
```

```
#:> array([1. , 1.2, 1.4, 1.6, 1.8, 2. , 2.2, 2.4, 2.6, 2.8])
```

12.6 Sampling (Integer)

```
random.choice( a, size=None, replace=True, p=None)  # a=integer, return <size> integer
random.choice( a, size=None, replace=True, p=None)  # a=array-
like, return <size> integers picked from list a
```

```
np.random.choice (100, size=10)
```

```
#:> array([58,  0, 84, 50, 89, 32, 87, 30, 66, 92])
```

```
np.random.choice( [1,3,5,7,9,11,13,15,17,19,21,23], size=10, replace=False)
```

```
#:> array([ 5,  1, 23, 17,  3, 13, 15,  9, 21,  7])
```

12.7 NaN : Missing Numerical Data

- You should be aware that NaN is a bit like a data virus?it infects any other object it touches

```
t = np.array([1, np.nan, 3, 4])
t.dtype
```

```
#:> dtype('float64')
```

Regardless of the operation, the result of arithmetic with NaN will be another NaN

```
1 + np.nan
```

```
#:> nan
```

```
t.sum(), t.mean(), t.max()
```

```
#:> (nan, nan, nan)
```

```
np.nansum(t), np.nanmean(t), np.nanmax(t)
```

```
#:> (8.0, 2.6666666666666665, 4.0)
```

Chapter 13

pandas

13.1 Modules Import

```
import pandas as pd

## Other Libraries
import numpy as np
import datetime as dt
from datetime import datetime
from datetime import date
```

13.2 Pandas Objects

13.2.1 Pandas Data Types

- pandas.Timestamp
- pandas.Timedelta
- pandas.Period
- pandas.Interval
- pandas.DateTimeIndex

13.2.2 Pandas Data Structure

Type	Dimension	Size	Value	Constructor
Series	1	Immutable	Mutable	pandas.DataFrame(data, index, dtype, copy)
DataFrame	2	Mutable	Mutable	pandas.DataFrame(data, index, columns, dtype, copy)

Type	Dimension	Size	Value	Constructor
Panel	3	Mutable	Mutable	

data can be ndarray, list, constants
index must be unique and same length as data. Can be integer or string **dtype** if none, it will be inferred
copy copy data. Default false

13.3 Class Method

13.3.1 Creating Timestamp Objects

Pandas **`to_datetime()`** can:
- Convert list of dates to **DateTimeIndex**
- Convert list of dates to **Series of Timestamps**
- Convert single date into **Timestamp** Object . Source can be **string, date, datetime object**

13.3.1.1 From List to `DateTimeIndex`

```
dti = pd.to_datetime(['2011-01-03',              # from string
                      date(2018,4,13),           # from date
                      datetime(2018,3,1,7,30)] # from datetime
              )
print( dti,
      '\nObject Type:  ', type(dti),
      '\nObject dtype: ', dti.dtype,
      '\nElement Type: ', type(dti[1]))
```

```
#:> DatetimeIndex(['2011-01-03 00:00:00', '2018-04-13 00:00:00', '2018-
03-01 07:30:00'], dtype='datetime64[ns]', freq=None)
#:> Object Type:   <class 'pandas.core.indexes.datetimes.DatetimeIndex'>
#:> Object dtype:  datetime64[ns]
#:> Element Type:  <class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

13.3.1.2 From List to Series of Timestamps

```
sdt = pd.to_datetime(pd.Series(['2011-01-03',      # from string
                                date(2018,4,13),        # from date
                                datetime(2018,3,1,7,30)]# from datetime
              ))
print(sdt,
```

```
      '\nObject Type:  ',type(sdt),
      '\nObject dtype: ', sdt.dtype,
      '\nElement Type: ',type(sdt[1]))
```

```
#:> 0   2011-01-03 00:00:00
#:> 1   2018-04-13 00:00:00
#:> 2   2018-03-01 07:30:00
#:> dtype: datetime64[ns]
#:> Object Type:   <class 'pandas.core.series.Series'>
#:> Object dtype:  datetime64[ns]
#:> Element Type:  <class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

13.3.1.3 From Scalar to Timestamp

```
print( pd.to_datetime('2011-01-03'), '\n',
       pd.to_datetime(date(2011,1,3)), '\n',
       pd.to_datetime(datetime(2011,1,3,5,30)), '\n',
       '\nElement Type: ', type(pd.to_datetime(datetime(2011,1,3,5,30))))
```

```
#:> 2011-01-03 00:00:00
#:>  2011-01-03 00:00:00
#:>  2011-01-03 05:30:00
#:>
#:> Element Type:  <class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

13.3.2 Generate Timestamp Sequence

The function `date_range()` return **`DateTimeIndex`** object. Use `Series()` to convert into Series if desired.

13.3.2.1 Hourly

If start time not specified, default to 00:00:00.
If start time specified, it will be honored on all subsequent Timestamp elements.
Specify **start** and **end**, sequence will automatically distribute Timestamp according to **frequency**.

```
print(
  pd.date_range('2018-01-01', periods=3, freq='H'),
  pd.date_range(datetime(2018,1,1,12,30), periods=3, freq='H'),
  pd.date_range(start='2018-01-03-1230', end='2018-01-03-18:30', freq='H'))
```

```
#:> DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 01:00:00', '2018-
01-01 02:00:00'], dtype='datetime64[ns]', freq='H') DatetimeIndex(['2018-
01-01 12:30:00', '2018-01-01 13:30:00', '2018-01-01 14:30:00'], dtype='datetime64[ns]', freq='H')
01-03 12:30:00', '2018-01-03 13:30:00', '2018-01-03 14:30:00',
```

```
#:>                 '2018-01-03 15:30:00', '2018-01-03 16:30:00', '2018-
01-03 17:30:00',
#:>                 '2018-01-03 18:30:00'],
#:>                dtype='datetime64[ns]', freq='H')
```

13.3.2.2 Daily

When the **frequency is Day and time is not specified**, output is date distributed.
When time is specified, output will honor the time.

```
print(
  pd.date_range(date(2018,1,2), periods=3, freq='D'),
  pd.date_range('2018-01-01-1230', periods=4, freq='D'))
```

```
#:> DatetimeIndex(['2018-01-02', '2018-01-03', '2018-01-04'], dtype='datetime64[ns]', 
01-01 12:30:00', '2018-01-02 12:30:00', '2018-01-03 12:30:00',
#:>                 '2018-01-04 12:30:00'],
#:>                dtype='datetime64[ns]', freq='D')
```

13.3.2.3 First Day Of Month

Use `freq=MS`, M stands for montly, S stand for Start. If the **day** specified, the sequence start from first day of following month.

```
print(
  pd.date_range('2018-01', periods=4, freq='MS'),
  pd.date_range('2018-01-09', periods=4, freq='MS'),
  pd.date_range('2018-01-09 12:30:00', periods=4, freq='MS') )
```

```
#:> DatetimeIndex(['2018-01-01', '2018-02-01', '2018-03-01', '2018-
04-01'], dtype='datetime64[ns]', freq='MS') DatetimeIndex(['2018-
02-01', '2018-03-01', '2018-04-01', '2018-05-01'], dtype='datetime64[ns]', freq='MS') 
02-01 12:30:00', '2018-03-01 12:30:00', '2018-04-01 12:30:00',
#:>                 '2018-05-01 12:30:00'],
#:>                dtype='datetime64[ns]', freq='MS')
```

13.3.2.4 Last Day of Month

Sequence always starts from the end of the specified month.

```
print(
  pd.date_range('2018-01', periods=4, freq='M'),
  pd.date_range('2018-01-09', periods=4, freq='M'),
  pd.date_range('2018-01-09 12:30:00', periods=4, freq='M'))
```

```
#:> DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-
04-30'], dtype='datetime64[ns]', freq='M') DatetimeIndex(['2018-
```

```
01-31', '2018-02-28', '2018-03-31', '2018-04-30'], dtype='datetime64[ns]', freq='M') DatetimeInde
01-31 12:30:00', '2018-02-28 12:30:00', '2018-03-31 12:30:00',
#:>                '2018-04-30 12:30:00'],
#:>               dtype='datetime64[ns]', freq='M')
```

13.3.3 Frequency Table (crosstab)

crosstab returns **Dataframe** Object

```
crosstab( index = <SeriesObj>, columns = <new_colName> )                # one dimension table
crosstab( index = <SeriesObj>, columns = <SeriesObj> )                  # two dimension table
crosstab( index = <SeriesObj>, columns = [<SeriesObj1>, <SeriesObj2>] ) # multi dimension table
crosstab( index = <SeriesObj>, columns = <SeriesObj>, margines=True )   # add column and row marg
```

13.3.3.1 Sample Data

```
n = 200
comp = ['C' + i for i in np.random.randint( 1,4, size  = n).astype(str)] # 3x Company
dept = ['D' + i for i in np.random.randint( 1,6, size  = n).astype(str)] # 5x Department
grp =  ['G' + i for i in np.random.randint( 1,3, size  = n).astype(str)] # 2x Groups
value1 = np.random.normal( loc=50 , scale=5 , size = n)
value2 = np.random.normal( loc=20 , scale=3 , size = n)
value3 = np.random.normal( loc=5 , scale=30 , size = n)

mydf = pd.DataFrame({
    'comp':comp,
    'dept':dept,
    'grp': grp,
    'value1':value1,
    'value2':value2,
    'value3':value3 })
mydf.head()
```

```
#:>   comp dept grp     value1     value2     value3
#:> 0   C2   D2  G1  45.546848  20.929492  -5.538823
#:> 1   C3   D2  G1  54.543313  23.236250  -8.551392
#:> 2   C1   D5  G1  56.130200  18.587278   1.741133
#:> 3   C2   D5  G2  48.368472  15.446616 -40.355283
#:> 4   C2   D2  G2  64.646426  20.088961  -3.323800
```

13.3.3.2 One DimensionTable

```
## Frequency Countn For Company, Department
print(
  pd.crosstab(index=mydf.comp, columns='counter'),'\n\n',
  pd.crosstab(index=mydf.dept, columns='counter'))
```

```
#:> col_0  counter
#:> comp
#:> C1          75
#:> C2          62
#:> C3          63
#:>
#:>  col_0  counter
#:> dept
#:> D1          33
#:> D2          43
#:> D3          35
#:> D4          52
#:> D5          37
```

13.3.3.3 Two Dimension Table

```
pd.crosstab(index=mydf.comp, columns=mydf.dept)
```

```
#:> dept  D1  D2  D3  D4  D5
#:> comp
#:> C1     9  18  12  21  15
#:> C2    10  10  18  15   9
#:> C3    14  15   5  16  13
```

13.3.3.4 Higher Dimension Table

Crosstab header is **multi-levels index** when more than one column specified.

```
tb = pd.crosstab(index=mydf.comp, columns=[mydf.dept, mydf.grp])
print( tb, '\n\n',
       tb.columns )
```

```
#:> dept D1     D2    D3     D4     D5
#:> grp  G1 G2  G1 G2 G1  G2 G1  G2 G1 G2
#:> comp
#:> C1    5  4  11  7  4   8  9  12  9  6
#:> C2    6  4   3  7  6  12  8   7  5  4
#:> C3    9  5   8  7  3   2  8   8  5  8
#:>
#:>  MultiIndex([('D1', 'G1'),
#:>              ('D1', 'G2'),
#:>              ('D2', 'G1'),
#:>              ('D2', 'G2'),
#:>              ('D3', 'G1'),
#:>              ('D3', 'G2'),
#:>              ('D4', 'G1'),
#:>              ('D4', 'G2'),
```

```
#:>              ('D5', 'G1'),
#:>              ('D5', 'G2')],
#:>             names=['dept', 'grp'])
```

Select **sub-dataframe** using multi-level referencing.

```
print( 'Under D2:\n', tb['D2'], '\n\n',
       'Under D2-G2:\n',tb['D2','G1'])
```

```
#:> Under D2:
#:>  grp   G1  G2
#:> comp
#:> C1    11   7
#:> C2     3   7
#:> C3     8   7
#:>
#:>  Under D2-G2:
#:>  comp
#:> C1    11
#:> C2     3
#:> C3     8
#:> Name: (D2, G1), dtype: int64
```

13.3.3.5 Getting Margin

Extend the crosstab with 'margin=True' to have sum of rows/columns, presented in **new column/row named 'All'**.

```
tb = pd.crosstab(index=mydf.dept, columns=mydf.grp, margins=True)
tb
```

```
#:> grp   G1   G2  All
#:> dept
#:> D1    20   13   33
#:> D2    22   21   43
#:> D3    13   22   35
#:> D4    25   27   52
#:> D5    19   18   37
#:> All   99  101  200
```

```
print(
  'Row Sums:     \n', tb.loc[:,'All'],
  '\n\nColumn Sums:\n', tb.loc['All'])
```

```
#:> Row Sums:
#:>  dept
#:> D1      33
#:> D2      43
#:> D3      35
```

```
#:> D4      52
#:> D5      37
#:> All    200
#:> Name: All, dtype: int64
#:>
#:> Column Sums:
#:>  grp
#:> G1      99
#:> G2     101
#:> All    200
#:> Name: All, dtype: int64
```

13.3.3.6 Getting Proportion

Use matrix operation divide each row with its respective column sum.

```
tb/tb.loc['All']
```

```
#:> grp         G1        G2    All
#:> dept
#:> D1    0.202020  0.128713  0.165
#:> D2    0.222222  0.207921  0.215
#:> D3    0.131313  0.217822  0.175
#:> D4    0.252525  0.267327  0.260
#:> D5    0.191919  0.178218  0.185
#:> All   1.000000  1.000000  1.000
```

13.3.4 Concatination

13.3.4.1 Sample Data

```
s1 = pd.Series(['A1','A2','A3','A4'])
s2 = pd.Series(['B1','B2','B3','B4'], name='B')
s3 = pd.Series(['C1','C2','C3','C4'], name='C')
```

13.3.4.2 Column-Wise

Combinining Multiple Series Into A New DataFrame
- Added series will have 0,1,2,... column names (if Series are not named originally)
- None series will be ignored
- `axis=1` means column-wise

```
pd.concat([s1,s2,s3, None], axis=1)
```

```
#:>     0   B   C
#:> 0  A1  B1  C1
#:> 1  A2  B2  C2
#:> 2  A3  B3  C3
```

```
#:> 3  A4  B4  C4
```

Add Multiple Series Into An Existing DataFrame
- No change to original data frame column name
- Added columns from series will have 0,1,2,3,.. column name

```
df = pd.DataFrame({ 'A': s1, 'B': s2})
pd.concat([df,s3,s1, None],axis=1)
```

```
#:>     A   B   C   0
#:> 0  A1  B1  C1  A1
#:> 1  A2  B2  C2  A2
#:> 2  A3  B3  C3  A3
#:> 3  A4  B4  C4  A4
```

13.3.4.3 Row-Wise

13.3.5 External Data

13.3.5.1 `html_table` Parser

This method require **html5lib** library.
- Read the web page, create a list: which contain one or more dataframes that maps to each html table found
- Scrap all detectable html tables
- Auto detect column header
- Auto create index using number starting from 0

```
read_html(url)  # return list of dataframe(s) that maps to web table(s) structure
```

```
df_list = pd.read_html('https://www.malaysiastock.biz/Listed-Companies.aspx?type=S&s1=18')  ## re
df = df_list[6]  ## get the specific table

print ('Total Table(s) Found : ', len(df_list), '\n',
       'First Table Found:      ',df)
```

```
#:> Total Table(s) Found :  11
#:>  First Table Found:                 0                                 1
#:> 0  Sector:  --- Filter by Sector ---  BOND ISLAMIC  CLOSED...
```

13.3.5.2 CSV Writing

Syntax

```
DataFrame.to_csv(
  path_or_buf=None,   ## if not provided, result is returned as string
  sep=', ',
  na_rep='',
  float_format=None,
  columns=None,       ## list of columns name to write, if not provided, all columns are written
```

```
  header=True,        ## write out column names
  index=True,         ## write row label
  index_label=None,
  mode='w',
  encoding=None,      ## if not provided, default to 'utf-8'
  quoting=None, quotechar='"',
  line_terminator=None,
  chunksize=None,
  date_format=None,
  doublequote=True,
  escapechar=None,
  decimal='.')
```

Example below shows column value containing different special character. Note that pandas handles these very well by default.

```
mydf = pd.DataFrame({'Id':[10,20,30,40],
                     'Name':  ['Aaa','Bbb','Ccc','Ddd'],
                     'Funny': ["world's most \clever",
                     "Bloody, damn, good",
                     "many\nmany\nline",
                     'Quoting "is" tough']})
mydf.set_index('Id', inplace=True)
mydf.to_csv('data/csv_test.csv', index=True)
mydf
```

```
#:>    Name                 Funny
#:> Id
#:> 10  Aaa  world's most \clever
#:> 20  Bbb    Bloody, damn, good
#:> 30  Ccc      many\nmany\nline
#:> 40  Ddd    Quoting "is" tough
```

This is the file saved

```
# system('more data\\csv_test.csv')
```

All content retained when reading back by Pandas

```
pd.read_csv('data/csv_test.csv', index_col='Id')
```

```
#:>    Name                 Funny
#:> Id
#:> 10  Aaa  world's most \clever
#:> 20  Bbb    Bloody, damn, good
#:> 30  Ccc      many\nmany\nline
#:> 40  Ddd    Quoting "is" tough
```

13.3.5.3 CSV Reading

Syntax

```
pandas.read_csv(
    'url or filePath',                     # path to file or url
    encoding    = 'utf_8',                 # optional: default is 'utf_8'
    index_col   = ['colName1', ...],       # optional: specify one or more index column
    parse_dates = ['dateCol1', ...],       # optional: specify multiple string column to convert
    na_values   = ['.','na','NA','N/A'],   # optional: values that is considered NA
    names       = ['newColName1', ... ],   # optional: overwrite column names
    thousands   = '.',                     # optional: thousand seperator symbol
    nrows       = n,                       # optional: load only first n rows
    skiprows    = 0,                       # optional: don't load first n rows
    parse_dates = False,                   # List of date column names
    infer_datetime_format = False          # automatically parse dates
)
```

Refer to full codec Python Codec.

Default Import

- index is sequence of integer 0,1,2…
- only two data types detection; **number (float64/int64) and string (object)**
- **date is not parsed**, hence stayed as string

```
goo = pd.read_csv('data/goog.csv', encoding='utf_8')
print(goo.head(), '\n\n',
      goo.info())
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 61 entries, 0 to 60
#:> Data columns (total 6 columns):
#:>  #   Column  Non-Null Count  Dtype
#:> ---  ------  --------------  -----
#:>  0   Date    61 non-null     object
#:>  1   Open    61 non-null     float64
#:>  2   High    61 non-null     float64
#:>  3   Low     61 non-null     float64
#:>  4   Close   61 non-null     float64
#:>  5   Volume  61 non-null     int64
#:> dtypes: float64(4), int64(1), object(1)
#:> memory usage: 3.0+ KB
#:>          Date        Open        High         Low       Close   Volume
#:> 0  12/19/2016  790.219971  797.659973  786.270020  794.200012  1225900
#:> 1  12/20/2016  796.760010  798.650024  793.270020  796.419983   925100
```

```
#:> 2  12/21/2016  795.840027  796.676025  787.099976  794.559998  1208700
#:> 3  12/22/2016  792.359985  793.320007  788.580017  791.260010   969100
#:> 4  12/23/2016  790.900024  792.739990  787.280029  789.909973   623400
#:> 
#:>  None
```

Specify Data Types

- To customize the data type, use **dtype** parameter with a **dict** of definition.

```
d_types = {'Volume': str}
pd.read_csv('data/goog.csv', dtype=d_types).info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 61 entries, 0 to 60
#:> Data columns (total 6 columns):
#:>  #   Column  Non-Null Count  Dtype  
#:> ---  ------  --------------  -----  
#:>  0   Date    61 non-null     object 
#:>  1   Open    61 non-null     float64
#:>  2   High    61 non-null     float64
#:>  3   Low     61 non-null     float64
#:>  4   Close   61 non-null     float64
#:>  5   Volume  61 non-null     object 
#:> dtypes: float64(4), object(2)
#:> memory usage: 3.0+ KB
```

Parse Datetime

You can specify multiple date-alike column for parsing

```
pd.read_csv('data/goog.csv', parse_dates=['Date']).info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 61 entries, 0 to 60
#:> Data columns (total 6 columns):
#:>  #   Column  Non-Null Count  Dtype         
#:> ---  ------  --------------  -----         
#:>  0   Date    61 non-null     datetime64[ns]
#:>  1   Open    61 non-null     float64       
#:>  2   High    61 non-null     float64       
#:>  3   Low     61 non-null     float64       
#:>  4   Close   61 non-null     float64       
#:>  5   Volume  61 non-null     int64         
#:> dtypes: datetime64[ns](1), float64(4), int64(1)
#:> memory usage: 3.0 KB
```

Parse Datetime, Then Set as Index

- Specify names of date column in `parse_dates=`

- When date is set as index, the type is **DateTimeIndex**

```
goo3 = pd.read_csv('data/goog.csv',index_col='Date', parse_dates=['Date'])
goo3.info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> DatetimeIndex: 61 entries, 2016-12-19 to 2017-03-17
#:> Data columns (total 5 columns):
#:>  #   Column  Non-Null Count  Dtype
#:> ---  ------  --------------  -----
#:>  0   Open    61 non-null     float64
#:>  1   High    61 non-null     float64
#:>  2   Low     61 non-null     float64
#:>  3   Close   61 non-null     float64
#:>  4   Volume  61 non-null     int64
#:> dtypes: float64(4), int64(1)
#:> memory usage: 2.9 KB
```

13.3.6 Inspection

13.3.6.1 Structure info

info() is a function that print information to screen. It doesn't return any object

```
dataframe.info()  # display columns and number of rows (that has no missing data)
```

```
goo.info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 61 entries, 0 to 60
#:> Data columns (total 6 columns):
#:>  #   Column  Non-Null Count  Dtype
#:> ---  ------  --------------  -----
#:>  0   Date    61 non-null     object
#:>  1   Open    61 non-null     float64
#:>  2   High    61 non-null     float64
#:>  3   Low     61 non-null     float64
#:>  4   Close   61 non-null     float64
#:>  5   Volume  61 non-null     int64
#:> dtypes: float64(4), int64(1), object(1)
#:> memory usage: 3.0+ KB
```

13.3.6.2 head

```
goo.head()
```

```
#:>          Date        Open        High         Low       Close   Volume
#:> 0  12/19/2016  790.219971  797.659973  786.270020  794.200012  1225900
```

```
#:> 1  12/20/2016  796.760010  798.650024  793.270020  796.419983   925100
#:> 2  12/21/2016  795.840027  796.676025  787.099976  794.559998  1208700
#:> 3  12/22/2016  792.359985  793.320007  788.580017  791.260010   969100
#:> 4  12/23/2016  790.900024  792.739990  787.280029  789.909973   623400
```

13.4 class: Timestamp

This is an enhanced version to datetime standard library.
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Timestamp.html#pandas.Timestamp

13.4.1 Constructor

13.4.1.1 From Number

```
print( pd.Timestamp(year=2017, month=1, day=1),'\n',  #date-like numbers
       pd.Timestamp(2017,1,1), '\n',                  # date-like numbers
       pd.Timestamp(2017,12,11,5,45),'\n',            # datetime-like numbers
       pd.Timestamp(2017,12,11,5,45,55,999),'\n',     # + microseconds
       pd.Timestamp(2017,12,11,5,45,55,999,8),'\n',   # + nanoseconds
       type(pd.Timestamp(2017,12,11,5,45,55,999,8)),'\n')
```

```
#:> 2017-01-01 00:00:00
#:>  2017-01-01 00:00:00
#:>  2017-12-11 05:45:00
#:>  2017-12-11 05:45:55.000999
#:>  2017-12-11 05:45:55.000999008
#:>  <class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

13.4.1.2 From String

Observe that pandas support many string input format

Year Month Day, default has no timezone

```
print( pd.Timestamp('2017-12-11'),'\n',   # date-like string: year-month-day
       pd.Timestamp('2017 12 11'),'\n',   # date-like string: year-month-day
       pd.Timestamp('2017 Dec 11'),'\n',  # date-like string: year-month-day
       pd.Timestamp('Dec 11, 2017'))      # date-like string: year-month-day
```

```
#:> 2017-12-11 00:00:00
#:>  2017-12-11 00:00:00
#:>  2017-12-11 00:00:00
#:>  2017-12-11 00:00:00
```

YMD Hour Minute Second Ms

```
print( pd.Timestamp('2017-12-11 0545'),'\n',     ## hour minute
       pd.Timestamp('2017-12-11-05:45'),'\n',
       pd.Timestamp('2017-12-11T0545'),'\n',
       pd.Timestamp('2017-12-11 054533'),'\n',   ## hour minute seconds
       pd.Timestamp('2017-12-11 05:45:33'))
```

```
#:> 2017-12-11 05:45:00
#:>  2017-12-11 05:45:00
#:>  2017-12-11 05:45:00
#:>  2017-12-11 05:45:33
#:>  2017-12-11 05:45:33
```

With Timezone can be included in various ways.

```
print( pd.Timestamp('2017-01-01T0545Z'),'\n',  # GMT
       pd.Timestamp('2017-01-01T0545+9'),'\n', # GMT+9
       pd.Timestamp('2017-01-01T0545+0800'),'\n',   # GMT+0800
       pd.Timestamp('2017-01-01 0545', tz='Asia/Singapore'),'\n')
```

```
#:> 2017-01-01 05:45:00+00:00
#:>  2017-01-01 05:45:00+09:00
#:>  2017-01-01 05:45:00+08:00
#:>  2017-01-01 05:45:00+08:00
```

13.4.1.3 From Standard Library datetime and date Object

```
print( pd.Timestamp(date(2017,3,5)),'\n',           # from date
       pd.Timestamp(datetime(2017,3,5,4,30)),'\n',  # from datetime
       pd.Timestamp(datetime(2017,3,5,4,30), tz='Asia/Kuala_Lumpur')) # from datetime, + tz
```

```
#:> 2017-03-05 00:00:00
#:>  2017-03-05 04:30:00
#:>  2017-03-05 04:30:00+08:00
```

13.4.2 Attributes

We can tell many things about a Timestamp object.

```
ts = pd.Timestamp('2017-01-01T054533+0800') # GMT+0800
print( ts.month, '\n',
       ts.day, '\n',
       ts.year, '\n',
       ts.hour, '\n',
       ts.minute, '\n',
       ts.second, '\n',
       ts.microsecond, '\n',
       ts.nanosecond, '\n',
```

```
       ts.tz, '\n',
       ts.daysinmonth,'\n',
       ts.dayofyear, '\n',
       ts.is_leap_year, '\n',
       ts.is_month_end, '\n',
       ts.is_month_start, '\n',
       ts.dayofweek)
```

```
#:> 1
#:>  1
#:>  2017
#:>  5
#:>  45
#:>  33
#:>  0
#:>  0
#:>  pytz.FixedOffset(480)
#:>  31
#:>  1
#:>  False
#:>  False
#:>  True
#:>  6
```

Note that timezone (tz) is a **pytz object**.

```
ts1 = pd.Timestamp(datetime(2017,3,5,4,30), tz='Asia/Kuala_Lumpur')   # from datetime,
ts2 = pd.Timestamp('2017-01-01T054533+0800') # GMT+0800
ts3 = pd.Timestamp('2017-01-01T0545')

print( ts1.tz, 'Type:', type(ts1.tz), '\n',
       ts2.tz, 'Type:', type(ts2.tz), '\n',
       ts3.tz, 'Type:', type(ts3.tz)  )
```

```
#:> Asia/Kuala_Lumpur Type: <class 'pytz.tzfile.Asia/Kuala_Lumpur'>
#:>  pytz.FixedOffset(480) Type: <class 'pytz._FixedOffset'>
#:>  None Type: <class 'NoneType'>
```

13.4.3 Instance Methods

13.4.3.1 Atribute-like Methods

```
ts = pd.Timestamp(2017,1,1)
print( ' Weekday:    ', ts.weekday(), '\n',
       'ISO Weekday:',  ts.isoweekday(), '\n',
       'Day Name:   ',  ts.day_name(), '\n',
```

```
       'ISO Calendar:',  ts.isocalendar()
       )
```

```
#:>  Weekday:     6
#:>  ISO Weekday: 7
#:>  Day Name:    Sunday
#:>  ISO Calendar: (2016, 52, 7)
```

13.4.3.2 Timezones

Adding Timezones and Clock Shifting

- `tz_localize` will add the timezone, however will not shift the clock.
- Once a timestamp had gotten a timezone, you can easily shift the clock to another timezone using `tz_convert()`

```
ts = pd.Timestamp(2017,1,10,10,34)        ## No timezone
ts1 = ts.tz_localize('Asia/Kuala_Lumpur')  ## Add timezone
ts2 = ts1.tz_convert('UTC')                 ## Convert timezone
print(' Origininal Timestamp           :', ts,  '\n',
      'Loacalized Timestamp (added TZ):', ts1, '\n',
      'Converted Timestamp (shifted)  :',ts2)
```

```
#:>  Origininal Timestamp           : 2017-01-10 10:34:00
#:>  Loacalized Timestamp (added TZ): 2017-01-10 10:34:00+08:00
#:>  Converted Timestamp (shifted)  : 2017-01-10 02:34:00+00:00
```

Removing Timezone

Just apply **None with `tz_localize`** to remove TZ infomration.

```
ts = pd.Timestamp(2017,1,10,10,34)        ## No timezone
ts = ts.tz_localize('Asia/Kuala_Lumpur')  ## Add timezone
ts = ts.tz_localize(None)                 ## Convert timezone
ts
```

```
#:> Timestamp('2017-01-10 10:34:00')
```

13.4.3.3 Formatting

`strftime`

Use **`strftime()`** to customize string format. For complete directive, see below: https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

```
ts = pd.Timestamp(2017,1,10,10,34)        ## No timezone
ts = ts.tz_localize('Asia/Kuala_Lumpur')  ## Add timezone
ts.strftime("%m/%d")
```

```
#:> '01/10'
```

isoformat

Use **isoformat()** to format ISO string (**without timezone**)

```
ts = pd.Timestamp(2017,1,10,10,34)
ts1 = ts.tz_localize('Asia/Kuala_Lumpur')
print( ' ISO Format without TZ:', ts.isoformat(), '\n',
       'ISO Format with TZ   :', ts1.isoformat())
```

```
#:>  ISO Format without TZ: 2017-01-10T10:34:00
#:>  ISO Format with TZ   : 2017-01-10T10:34:00+08:00
```

13.4.3.4 Type Conversion

Convert To datetime.datetime/date

Use `to_pydatetime()` to convert into standard library **datetime.datetime**. From the 'datetime' object, apply `date()` to get **datetime.date**

```
ts = pd.Timestamp(2017,1,10,7,30,52)
print(
  'Datetime:',  ts.to_pydatetime(), '\n',
  'Date Only:', ts.to_pydatetime().date())
```

```
#:> Datetime: 2017-01-10 07:30:52
#:>  Date Only: 2017-01-10
```

Convert To numpy.datetime64

Use `to_datetime64()` to convert into `numpy.datetime64`

```
ts = pd.Timestamp(2017,1,10,7,30,52)
ts.to_datetime64()
```

```
#:> numpy.datetime64('2017-01-10T07:30:52.000000000')
```

13.4.3.5 ceil

```
print( ts.ceil(freq='D') ) # ceiling to day
```

```
#:> 2017-01-11 00:00:00
```

13.4.3.6 Updating

`replace()`

```
ts.replace(year=2000, month=1,day=1)
```

```
#:> Timestamp('2000-01-01 07:30:52')
```

13.5 class: DateTimeIndex

13.5.1 Creating

Refer to Pandas class method above.

13.5.2 Instance Method

13.5.2.1 Data Type Conversion

Convert To datetime.datetime
Use **`to_pydatetime`** to convert into python standard **datetime.datetime** object

```
print('Converted to List:', dti.to_pydatetime(), '\n\n',
      'Converted Type:',    type(dti.to_pydatetime()))
```

```
#:> Converted to List: [datetime.datetime(2011, 1, 3, 0, 0) datetime.datetime(2018, 4, 13, 0, 0)
#:>  datetime.datetime(2018, 3, 1, 7, 30)]
#:>
#:>  Converted Type: <class 'numpy.ndarray'>
```

13.5.2.2 Structure Conversion

Convert To Series: `to_series`
This creates a Series where **index and data** with the same value

```
#dti = pd.date_range('2018-02', periods=4, freq='M')
dti.to_series()
```

```
#:> 2011-01-03 00:00:00   2011-01-03 00:00:00
#:> 2018-04-13 00:00:00   2018-04-13 00:00:00
#:> 2018-03-01 07:30:00   2018-03-01 07:30:00
#:> dtype: datetime64[ns]
```

Convert To DataFrame: `to_frame()`
This convert to **single column DataFrame** with index as the same value

```
dti.to_frame()
```

```
#:>                                       0
#:> 2011-01-03 00:00:00 2011-01-03 00:00:00
#:> 2018-04-13 00:00:00 2018-04-13 00:00:00
#:> 2018-03-01 07:30:00 2018-03-01 07:30:00
```

13.5.3 Attributes

All Timestamp Attributes can be used upon DateTimeIndex.

```
print( dti.weekday, '\n',
       dti.month, '\n',
       dti.daysinmonth)
```

```
#:> Int64Index([0, 4, 3], dtype='int64')
#:>  Int64Index([1, 4, 3], dtype='int64')
#:>  Int64Index([31, 30, 31], dtype='int64')
```

13.6 class: Series

Series allows different data types (object class) as its element

```
pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
- data array-like, iterable, dict or scalar
- If dtype not specified, it will infer from data.
```

13.6.1 Constructor

13.6.1.1 Empty Series

Passing no data to constructor will result in empty series. By default, empty series dtype is float.

```
s = pd.Series(dtype='object')
print (s, '\n',
       type(s))
```

```
#:> Series([], dtype: object)
#:>  <class 'pandas.core.series.Series'>
```

13.6.1.2 From Scalar

If data is a scalar value, an **index must be provided**. The **value will be repeated** to match the length of index

```
pd.Series( 99, index = ['a','b','c','d'])
```

```
#:> a    99
#:> b    99
#:> c    99
#:> d    99
#:> dtype: int64
```

13.6.1.3 From array-like

From list

```
pd.Series(['a','b','c','d','e'])          # from Python list
```

```
#:> 0    a
#:> 1    b
#:> 2    c
#:> 3    d
#:> 4    e
#:> dtype: object
```

From numpy.array
If index is not specified, default to 0 and continue incrementally

```
pd.Series(np.array(['a','b','c','d','e']))
```

```
#:> 0    a
#:> 1    b
#:> 2    c
#:> 3    d
#:> 4    e
#:> dtype: object
```

From DateTimeIndex

```
pd.Series(pd.date_range('2011-1-1','2011-1-3'))
```

```
#:> 0   2011-01-01
#:> 1   2011-01-02
#:> 2   2011-01-03
#:> dtype: datetime64[ns]
```

13.6.1.4 From Dictionary

The **dictionary key** will be the index. Order is **not sorted**.

```
pd.Series({'a' : 0., 'c' : 5., 'b' : 2.})
```

```
#:> a    0.0
#:> c    5.0
#:> b    2.0
#:> dtype: float64
```

If **index sequence** is specifeid, then Series will forllow the index order
Objerve that **missing data** (index without value) will be marked as NaN

```
pd.Series({'a' : 0., 'c' : 1., 'b' : 2.},index = ['a','b','c','d'])
```

```
#:> a    0.0
#:> b    2.0
#:> c    1.0
#:> d    NaN
```

```
#:> dtype: float64
```

13.6.1.5 Specify Index

```
pd.Series(['a','b','c','d','e'], index=[10,20,30,40,50])
```

```
#:> 10    a
#:> 20    b
#:> 30    c
#:> 40    d
#:> 50    e
#:> dtype: object
```

13.6.1.6 Mix Element Types

dType will be **'object'** when there were mixture of classes

```
ser = pd.Series(['a',1,2,3])
print('Object Type :  ', type(ser),'\n',
      'Object dType:  ', ser.dtype,'\n',
      'Element 1 Type: ',type(ser[0]),'\n',
      'Elmeent 2 Type: ',type(ser[1]))
```

```
#:> Object Type :   <class 'pandas.core.series.Series'>
#:>  Object dType:   object
#:>  Element 1 Type:  <class 'str'>
#:>  Elmeent 2 Type:  <class 'int'>
```

13.6.1.7 Specify Data Types

By default, dtype is **inferred** from data.

```
ser1 = pd.Series([1,2,3])
ser2 = pd.Series([1,2,3], dtype="int8")
ser3 = pd.Series([1,2,3], dtype="object")

print(' Inferred:        ',ser1.dtype, '\n',
      'Specified int8:  ',ser2.dtype, '\n',
      'Specified object:',ser3.dtype)
```

```
#:>  Inferred:         int64
#:>  Specified int8:   int8
#:>  Specified object: object
```

13.6.2 Accessing Series

```
series     ( single/list/range_of_row_label/number ) # can cause confusion
series.loc ( single/list/range_of_row_label )
```

```
series.iloc( single/list/range_of_row_number )
```

13.6.2.1 Sample Data

```
s = pd.Series([1,2,3,4,5],index=['a','b','c','d','e'])
s
```

```
#:> a    1
#:> b    2
#:> c    3
#:> d    4
#:> e    5
#:> dtype: int64
```

13.6.2.2 by Row Number(s)

Single Item. Notice that inputing a number and list of number give different result.

```
print( 'Referencing by number:',s.iloc[1],'\n\n',
       '\nReferencing by list of number:\n',s.iloc[[1]])
```

```
#:> Referencing by number: 2
#:>
#:>
#:> Referencing by list of number:
#:>  b    2
#:> dtype: int64
```

Multiple Items

```
s.iloc[[1,3]]
```

```
#:> b    2
#:> d    4
#:> dtype: int64
```

Range (First 3)

```
s.iloc[:3]
```

```
#:> a    1
#:> b    2
#:> c    3
#:> dtype: int64
```

Range (Last 3)

```
s.iloc[-3:]
```

```
#:> c    3
```

```
#:> d    4
#:> e    5
#:> dtype: int64
```

Range (in between)

```
s.iloc[2:3]
```

```
#:> c    3
#:> dtype: int64
```

13.6.2.3 by Index(es)

Single Label. Notice the difference referencing input: single index and list of index.
Warning: if index is invalid, this will result in error.

```
print( s.loc['c'], '\n',
       s[['c']])
```

```
#:> 3
#:>  c    3
#:> dtype: int64
```

Multiple Labels

If index is not found, it will return **NaN**

```
# error: missing labels no longer supported
s.loc[['k','c']]
```

** Range of Labels **

```
s.loc['b':'d']
```

```
#:> b    2
#:> c    3
#:> d    4
#:> dtype: int64
```

13.6.2.4 Filtering

Use **logical array** to filter

```
s = pd.Series(range(1,8))
s[s<5]
```

```
#:> 0    1
#:> 1    2
#:> 2    3
#:> 3    4
#:> dtype: int64
```

Use **where**
The where method is an application of the if-then idiom. For each element in the calling Series, if `cond` is True the element is used; otherwise `other` is used.

```
.where(cond, other=nan, inplace=False)
```

```
print(s.where(s<4),'\n\n',
      s.where(s<4,other=None) )
```

```
#:> 0    1.0
#:> 1    2.0
#:> 2    3.0
#:> 3    NaN
#:> 4    NaN
#:> 5    NaN
#:> 6    NaN
#:> dtype: float64
#:>
#:>  0       1
#:> 1       2
#:> 2       3
#:> 3    None
#:> 4    None
#:> 5    None
#:> 6    None
#:> dtype: object
```

13.6.3 Updating Series

13.6.3.1 by Row Number(s)

```
s = pd.Series(range(1,7), index=['a','b','c','d','e','f'])
s[2] = 999
s[[3,4]] = 888,777
s
```

```
#:> a      1
#:> b      2
#:> c    999
#:> d    888
#:> e    777
#:> f      6
#:> dtype: int64
```

13.6.3.2 by Index(es)

```
s = pd.Series(range(1,7), index=['a','b','c','d','e','f'])
s['e'] = 888
s[['c','d']] = 777,888
s
```

```
#:> a      1
#:> b      2
#:> c    777
#:> d    888
#:> e    888
#:> f      6
#:> dtype: int64
```

13.6.4 Series Attributes

13.6.4.1 The Data

```
s = pd.Series([1,2,3,4,5],index=['a','b','c','d','e'],name='SuperHero')
s
```

```
#:> a    1
#:> b    2
#:> c    3
#:> d    4
#:> e    5
#:> Name: SuperHero, dtype: int64
```

13.6.4.2 The Attributes

```
print( ' Series Index:    ',s.index, '\n',
       'Series dType:    ', s.dtype, '\n',
       'Series Size:     ', s.size, '\n',
       'Series Shape:    ', s.shape, '\n',
       'Series Dimension:', s.ndim)
```

```
#:>  Series Index:     Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
#:>  Series dType:     int64
#:>  Series Size:      5
#:>  Series Shape:     (5,)
#:>  Series Dimension: 1
```

13.6.5 Instance Methods

13.6.5.1 Index Manipulation

.rename_axis()

```
s.rename_axis('haribulan')
```

```
#:> haribulan
#:> a    1
#:> b    2
#:> c    3
#:> d    4
#:> e    5
#:> Name: SuperHero, dtype: int64
```

`.reset_index()`

Resetting index will:
- Convert index to a normal column, with column named as **'index'**
- Index renumbered to 1,2,3
- Return **DataFrame** (became two columns)

```
s.reset_index()
```

```
#:>   index  SuperHero
#:> 0     a          1
#:> 1     b          2
#:> 2     c          3
#:> 3     d          4
#:> 4     e          5
```

13.6.5.2 Structure Conversion

- A series structure contain **`value`** (in numpy array), its **`dtype`** (data type of the numpy array).

- Use **`values`** to retrieve into '`numpy.ndarray`. Use **`dtype`** to understand the data type.

```
s = pd.Series([1,2,3,4,5])
print(' Series value:      ', s.values, '\n',
      'Series value type: ',  type(s.values), '\n',
      'Series dtype:      ',  s.dtype)
```

```
#:>  Series value:       [1 2 3 4 5]
#:>  Series value type:  <class 'numpy.ndarray'>
#:>  Series dtype:       int64
```

Convert To List using **`.tolist()`**

```
pd.Series.tolist(s)
```

```
#:> [1, 2, 3, 4, 5]
```

13.6.5.3 DataType Conversion

Use **astype()** to convert to another numpy supproted datatypes, results in a new Series.
Warning: casting to incompatible type will result in **error**

```
s.astype('int8')
```

```
#:> 0    1
#:> 1    2
#:> 2    3
#:> 3    4
#:> 4    5
#:> dtype: int8
```

13.6.6 Series Operators

The result of applying operator (arithmetic or logic) to Series object **returns a new Series object**

13.6.6.1 Arithmetic Operator

```
s1 = pd.Series( [100,200,300,400,500] )
s2 = pd.Series( [10, 20, 30, 40, 50] )
```

Apply To One Series Object

```
s1 - 100
```

```
#:> 0      0
#:> 1    100
#:> 2    200
#:> 3    300
#:> 4    400
#:> dtype: int64
```

Apply To Two Series Objects

```
s1 - s2
```

```
#:> 0     90
#:> 1    180
#:> 2    270
#:> 3    360
#:> 4    450
#:> dtype: int64
```

13.6.6.2 Logic Operator

- Apply logic operator to a Series return a **new Series** of boolean result
- This can be used for **Series or DataFrame filtering**

```
bs = pd.Series(range(0,10))
bs>3
```

```
#:> 0    False
#:> 1    False
#:> 2    False
#:> 3    False
#:> 4     True
#:> 5     True
#:> 6     True
#:> 7     True
#:> 8     True
#:> 9     True
#:> dtype: bool
```

```
~((bs>3) & (bs<8) | (bs>7))
```

```
#:> 0     True
#:> 1     True
#:> 2     True
#:> 3     True
#:> 4    False
#:> 5    False
#:> 6    False
#:> 7    False
#:> 8    False
#:> 9    False
#:> dtype: bool
```

13.6.7 Series `.str` Accesor

If the underlying data is **str** type, then pandas exposed various properties and methos through **`str` accessor**.

`SeriesObj.str.operatorFunction()`

Available Functions

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas str methods that mirror Python string methods:

len() lower() translate() islower() ljust() upper() startswith() isupper() rjust() find() endswith() isnumeric() center() rfind() isalnum() isdecimal() zfill() index()

isalpha() split() strip() rindex() isdigit() rsplit() rstrip() capitalize() isspace() partition() lstrip() swapcase() istitle() rpartition()

13.6.7.1 Regex Extractor

Extract capture **groups** in the regex pattern, by default in DataFrame (`expand=True`).

```
Series.str.extract(self, pat, flags=0, expand=True)
- expand=True: if result is single column, make it a Series instead of Dataframe.
```

```
s = pd.Series(['a1', 'b2', 'c3'])
print(
  ' Extracted Dataframe:\n', s.str.extract(r'([ab])(\d)'),'\n\n',
  'Extracted Dataframe witn Names:\n', s.str.extract(r'(?P<Letter>[ab])(\d)'))
```

```
#:>  Extracted Dataframe:
#:>       0    1
#:> 0    a    1
#:> 1    b    2
#:> 2  NaN  NaN
#:>
#:>  Extracted Dataframe witn Names:
#:>    Letter    1
#:> 0      a    1
#:> 1      b    2
#:> 2    NaN  NaN
```

Below ouptut single columne, use **`expand=False`** to make the result a **Series**, instead of DataFrame.

```
r = s.str.extract(r'[ab](\d)', expand=False)
print( r, '\n\n', type(r) )
```

```
#:> 0      1
#:> 1      2
#:> 2    NaN
#:> dtype: object
#:>
#:>  <class 'pandas.core.series.Series'>
```

13.6.7.2 Character Extractor

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
monte
```

```
#:> 0    Graham Chapman
#:> 1       John Cleese
```

```
#:> 2     Terry Gilliam
#:> 3         Eric Idle
#:> 4       Terry Jones
#:> 5     Michael Palin
#:> dtype: object
```

startwith

```
monte.str.startswith('T')
```

```
#:> 0    False
#:> 1    False
#:> 2     True
#:> 3    False
#:> 4     True
#:> 5    False
#:> dtype: bool
```

Slicing

```
monte.str[0:3]
```

```
#:> 0    Gra
#:> 1    Joh
#:> 2    Ter
#:> 3    Eri
#:> 4    Ter
#:> 5    Mic
#:> dtype: object
```

13.6.7.3 Splitting

Split strings around given separator/delimiter in either string or regex.

```
Series.str.split(self, pat=None, n=-1, expand=False)
- pat: can be string or regex
```

```
s = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h_i_j'])
s
```

```
#:> 0        a_b_c
#:> 1        c_d_e
#:> 2          NaN
#:> 3    f_g_h_i_j
#:> dtype: object
```

str.split() by default, split will split each item into **array**

```
s.str.split('_')
```

```
#:> 0          [a, b, c]
```

```
#:> 1          [c, d, e]
#:> 2                NaN
#:> 3    [f, g, h, i, j]
#:> dtype: object
```

expand=True will return a **dataframe** instead of series. By default, expand split into all possible columns.

```
print( s.str.split('_', expand=True) )
```

```
#:>      0    1    2     3     4
#:> 0    a    b    c  None  None
#:> 1    c    d    e  None  None
#:> 2  NaN  NaN  NaN   NaN   NaN
#:> 3    f    g    h     i     j
```

It is possible to limit the number of columns splitted

```
print( s.str.split('_', expand=True, n=1) )
```

```
#:>      0        1
#:> 0    a      b_c
#:> 1    c      d_e
#:> 2  NaN      NaN
#:> 3    f  g_h_i_j
```

str.rsplit()

rsplit stands for **reverse split**, it works the same way, except it is reversed

```
print( s.str.rsplit('_', expand=True, n=1) )
```

```
#:>          0    1
#:> 0      a_b    c
#:> 1      c_d    e
#:> 2      NaN  NaN
#:> 3  f_g_h_i    j
```

13.6.7.4 Case Conversion

```
SeriesObj.str.upper()
SeriesObj.str.lower()
SeriesObj.str.capitalize()
```

```
s = pd.Series(['A', 'B', 'C', 'aAba', 'bBaca', np.nan, 'cCABA', 'dog', 'cat'])
print( s.str.upper(), '\n',
       s.str.capitalize())
```

```
#:> 0        A
#:> 1        B
#:> 2        C
```

```
#:> 3     AABA
#:> 4    BBACA
#:> 5      NaN
#:> 6    CCABA
#:> 7      DOG
#:> 8      CAT
#:> dtype: object
#:>  0        A
#:> 1        B
#:> 2        C
#:> 3     Aaba
#:> 4    Bbaca
#:> 5      NaN
#:> 6    Ccaba
#:> 7      Dog
#:> 8      Cat
#:> dtype: object
```

13.6.7.5 Number of Characters

```
s.str.len()
```

```
#:> 0    1.0
#:> 1    1.0
#:> 2    1.0
#:> 3    4.0
#:> 4    5.0
#:> 5    NaN
#:> 6    5.0
#:> 7    3.0
#:> 8    3.0
#:> dtype: float64
```

13.6.7.6 String Indexing

This return specified character from each item.

```
s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,'CABA', 'dog', 'cat'])
s.str[0].values    # first char
```

```
#:> array(['A', 'B', 'C', 'A', 'B', nan, 'C', 'd', 'c'], dtype=object)
```

```
s.str[0:2].values  # first and second char
```

```
#:> array(['A', 'B', 'C', 'Aa', 'Ba', nan, 'CA', 'do', 'ca'], dtype=object)
```

13.6.7.7 Series Substring Extraction

Sample Data

```
s = pd.Series(['a1', 'b2', 'c3'])
s
```

```
#:> 0    a1
#:> 1    b2
#:> 2    c3
#:> dtype: object
```

Extract absed on regex matching
… to improve …

```
type(s.str.extract('([ab])(\d)', expand=False))
```

```
#:> <class 'pandas.core.frame.DataFrame'>
```

13.6.8 Series `.dt` Accessor

If the underlying data is **datetime64** type, then pandas exposed various properties and methos through **`dt` accessor**.

13.6.8.1 Sample Data

```
s = pd.Series([
    datetime(2000,1,1,0,0,0),
    datetime(1999,12,15,12,34,55),
    datetime(2020,3,8,5,7,12),
    datetime(2018,1,1,0,0,0),
    datetime(2003,3,4,5,6,7)
])
s
```

```
#:> 0   2000-01-01 00:00:00
#:> 1   1999-12-15 12:34:55
#:> 2   2020-03-08 05:07:12
#:> 3   2018-01-01 00:00:00
#:> 4   2003-03-04 05:06:07
#:> dtype: datetime64[ns]
```

13.6.8.2 Convert To

datetime.datetime
Use **`to_pydatetime()`** to convert into **`numpy.array`** of standard library **`datetime.datetime`**

```
pdt  = s.dt.to_pydatetime()
print( type(pdt) )
```

```
#:> <class 'numpy.ndarray'>
```

```
pdt
```

```
#:> array([datetime.datetime(2000, 1, 1, 0, 0),
#:>        datetime.datetime(1999, 12, 15, 12, 34, 55),
#:>        datetime.datetime(2020, 3, 8, 5, 7, 12),
#:>        datetime.datetime(2018, 1, 1, 0, 0),
#:>        datetime.datetime(2003, 3, 4, 5, 6, 7)], dtype=object)
```

datetime.date
Use **`dt.date`** to convert into **`pandas.Series`** of standard library **`datetime.date`**
Is it possible to have a pandas.Series of datetime.datetime ? No, because Pandas want it as its own Timestamp.

```
sdt = s.dt.date
print( type(sdt[1] ))
```

```
#:> <class 'datetime.date'>
```

```
print( type(sdt))
```

```
#:> <class 'pandas.core.series.Series'>
```

```
sdt
```

```
#:> 0    2000-01-01
#:> 1    1999-12-15
#:> 2    2020-03-08
#:> 3    2018-01-01
#:> 4    2003-03-04
#:> dtype: object
```

13.6.8.3 Timestamp Attributes

A Series::DateTime object support below properties:
- date
- month
- day
- year
- dayofweek
- dayofyear
- weekday
- weekday_name
- quarter
- daysinmonth

- hour - minute

Full list below:
https://pandas.pydata.org/pandas-docs/stable/reference/series.html#datetimelike-properties

```
s.dt.date
```

```
#:> 0    2000-01-01
#:> 1    1999-12-15
#:> 2    2020-03-08
#:> 3    2018-01-01
#:> 4    2003-03-04
#:> dtype: object
```

```
s.dt.month
```

```
#:> 0     1
#:> 1    12
#:> 2     3
#:> 3     1
#:> 4     3
#:> dtype: int64
```

```
s.dt.dayofweek
```

```
#:> 0    5
#:> 1    2
#:> 2    6
#:> 3    0
#:> 4    1
#:> dtype: int64
```

```
s.dt.weekday
```

```
#:> 0    5
#:> 1    2
#:> 2    6
#:> 3    0
#:> 4    1
#:> dtype: int64
```

```
# error no attribute weekday_name
s.dt.weekday_name
```

```
s.dt.quarter
```

```
#:> 0    1
#:> 1    4
#:> 2    1
#:> 3    1
```

```
#:> 4    1
#:> dtype: int64
```

```
s.dt.daysinmonth
```

```
#:> 0    31
#:> 1    31
#:> 2    31
#:> 3    31
#:> 4    31
#:> dtype: int64
```

```
s.dt.time   # extract time as time Object
```

```
#:> 0    00:00:00
#:> 1    12:34:55
#:> 2    05:07:12
#:> 3    00:00:00
#:> 4    05:06:07
#:> dtype: object
```

```
s.dt.hour  # extract hour as integer
```

```
#:> 0     0
#:> 1    12
#:> 2     5
#:> 3     0
#:> 4     5
#:> dtype: int64
```

```
s.dt.minute # extract minute as integer
```

```
#:> 0     0
#:> 1    34
#:> 2     7
#:> 3     0
#:> 4     6
#:> dtype: int64
```

13.7 class: DataFrame

13.7.1 Constructor

13.7.1.1 Empty DataFrame

By default, An empty dataframe contain no coumns and index.

```
empty_df1 = pd.DataFrame()
empty_df2 = pd.DataFrame()
```

```
print(id(empty_df1), id(empty_df2), empty_df1)
```

```
#:> 140322763537552 140322721606736 Empty DataFrame
#:> Columns: []
#:> Index: []
```

However, you can also initialize an empty DataFrame with Index and/or Columns.

```
empty_df = pd.DataFrame(columns=['A','B','C'], index=[1,2,3])
print( empty_df )
```

```
#:>      A    B    C
#:> 1  NaN  NaN  NaN
#:> 2  NaN  NaN  NaN
#:> 3  NaN  NaN  NaN
```

Take note that below empty_df1 and empty_df2 refers to **same memory location**. Meaning they cantain similar data.

```
empty_df1 = empty_df2 = pd.DataFrame()
print(id(empty_df1), id(empty_df2))
```

```
#:> 140322721652176 140322721652176
```

13.7.1.2 From Row Oriented Data (List of Lists)

Create from **List of Lists**

```
DataFrame( [row_list1, row_list2, row_list3] )
DataFrame( [row_list1, row_list2, row_list3], column = columnName_list )
DataFrame( [row_list1, row_list2, row_list3], index  = row_label_list )
```

Basic DataFrame with default Row Label and Column Header

```
pd.DataFrame ([[101,'Alice',40000,2017],
               [102,'Bob',  24000, 2017],
               [103,'Charles',31000,2017]] )
```

```
#:>      0        1      2     3
#:> 0  101    Alice  40000  2017
#:> 1  102      Bob  24000  2017
#:> 2  103  Charles  31000  2017
```

Specify Column Header during Creation

```
pd.DataFrame ([[101,'Alice',40000,2017],
               [102,'Bob',  24000, 2017],
               [103,'Charles',31000,2017]], columns = ['empID','name','salary','year'])
```

```
#:>    empID     name  salary  year
#:> 0    101    Alice   40000  2017
```

```
#:> 1    102      Bob   24000  2017
#:> 2    103  Charles   31000  2017
```

Specify Row Label during Creation

```
pd.DataFrame ([[101,'Alice',40000,2017],
               [102,'Bob',  24000, 2017],
               [103,'Charles',31000,2017]], index   = ['r1','r2','r3'] )
```

```
#:>       0        1      2     3
#:> r1  101    Alice  40000  2017
#:> r2  102      Bob  24000  2017
#:> r3  103  Charles  31000  2017
```

13.7.1.3 From Row Oriented Data (List of Dictionary)

```
DataFrame( [dict1, dict2, dict3] )
DataFrame( [row_list1, row_list2, row_list3], column=np.arrange )
DataFrame( [row_list1, row_list2, row_list3], index=row_label_list )

by default,keys will become collumn names, and autosorted
```

Default Column Name Follow Dictionary Key
Note missing info as NaN

```
pd.DataFrame ([{"name":"Yong", "id":1,"zkey":101},{"name":"Gan","id":2}])
```

```
#:>    name  id   zkey
#:> 0  Yong   1  101.0
#:> 1   Gan   2    NaN
```

Specify Index

```
pd.DataFrame ([{"name":"Yong", "id":'wd1'},{"name":"Gan","id":'wd2'}],
             index = (1,2))
```

```
#:>    name   id
#:> 1  Yong  wd1
#:> 2   Gan  wd2
```

Specify Column Header during Creation, can acts as column filter and manual arrangement
Note missing info as NaN

```
pd.DataFrame ([{"name":"Yong", "id":1, "zkey":101},{"name":"Gan","id":2}],
              columns=("name","id","zkey"))
```

```
#:>    name  id   zkey
#:> 0  Yong   1  101.0
#:> 1   Gan   2    NaN
```

13.7.1.4 From Column Oriented Data

Create from **Dictrionary of List**

```
DataFrame(  { 'column1': list1,
              'column2': list2,
              'column3': list3 } ,
              index    = row_label_list,
              columns  = column_list)
```

By default, DataFrame will **arrange the columns alphabetically**, unless **columns** is specified

Default Row Label

```
data = {'empID':  [100,      101,    102,      103,     104],
        'year':   [2017,     2017,   2017,      2018,    2018],
        'salary': [40000,    24000,  31000,     20000,   30000],
        'name':   ['Alice', 'Bob',  'Charles', 'David', 'Eric']}
pd.DataFrame(data)
```

```
#:>    empID  year  salary     name
#:> 0    100  2017   40000    Alice
#:> 1    101  2017   24000      Bob
#:> 2    102  2017   31000  Charles
#:> 3    103  2018   20000    David
#:> 4    104  2018   30000     Eric
```

Specify Row Label during Creation

```
data = {'empID':  [100,      101,    102,      103,     104],
        'name':   ['Alice', 'Bob',  'Charles', 'David', 'Eric'],
        'year':   [2017,     2017,   2017,      2018,    2018],
        'salary': [40000,    24000,  31000,     20000,   30000] }
pd.DataFrame (data, index=['r1','r2','r3','r4','r5'])
```

```
#:>     empID     name  year  salary
#:> r1    100    Alice  2017   40000
#:> r2    101      Bob  2017   24000
#:> r3    102  Charles  2017   31000
#:> r4    103    David  2018   20000
#:> r5    104     Eric  2018   30000
```

Manualy Choose Columns and Arrangement

```
data = {'empID':  [100,      101,    102,      103,     104],
        'name':   ['Alice', 'Bob',  'Charles', 'David', 'Eric'],
        'year':   [2017,     2017,   2017,      2018,    2018],
        'salary': [40000,    24000,  31000,     20000,   30000] }
```

```
pd.DataFrame (data, columns=('empID','name','salary'), index=['r1','r2','r3','r4','r5'])
```

```
#:>     empID     name  salary
#:> r1    100    Alice   40000
#:> r2    101      Bob   24000
#:> r3    102  Charles   31000
#:> r4    103    David   20000
#:> r5    104     Eric   30000
```

13.7.2 Operator

13.7.2.1 The Data

Two dataframe is created, each with 3 columns and 3 rows. However, only two **matching column and row** names We shall notice that the operator will perform cell-wise, **honoring the row/column name**.

```
df1 = pd.DataFrame(data=
  {'idx': ['row1','row2','row3'],
   'x': [10, 20, 30],
   'y': [1,2,3],
   'z': [0.1, 0.2, 0.3]}).set_index('idx')

df2 = pd.DataFrame(data=
  {'idx': ['row1','row2','row4'],
   'x': [13, 23, 33],
   'z': [0.1, 0.2, 0.3],
   'k': [11,21,31]
   }).set_index('idx')

print( df1, '\n\n', df2)
```

```
#:>        x  y    z
#:> idx
#:> row1  10  1  0.1
#:> row2  20  2  0.2
#:> row3  30  3  0.3
#:>
#:>         x    z   k
#:> idx
#:> row1  13  0.1  11
#:> row2  23  0.2  21
#:> row4  33  0.3  31
```

13.7.2.2 Addition

Adding Two DataFrame

Using `+` operator, non-matching row/column names will result in **NA**. However, when using function **`add`**, none matching cells can be assumed as with a value.

```
r1 = df1 + df2
r2 = df1.add(df2,fill_value=1000)

print( r1, '\n\n', r2)
```

```
#:>        k     x   y    z
#:> idx
#:> row1 NaN  23.0 NaN  0.2
#:> row2 NaN  43.0 NaN  0.4
#:> row3 NaN   NaN NaN  NaN
#:> row4 NaN   NaN NaN  NaN
#:>
#:>             k       x       y       z
#:> idx
#:> row1  1011.0    23.0  1001.0     0.2
#:> row2  1021.0    43.0  1002.0     0.4
#:> row3     NaN  1030.0  1003.0  1000.3
#:> row4  1031.0  1033.0     NaN  1000.3
```

Adding Series and DataFrame

Specify the **appropriate `axis`** depending on the orientation of the series data. Column and Row names are respected in this operation. However, `fill_value` is **not applicable** when apply on Series.

Note that columns in Series that are not found in dataframe, will still be created in the result. This is similar behaviour as operating Dataframe with Dataframe.

```
s3 = pd.Series([1,1,1], index=['row1','row2','row4'])
s4 = pd.Series([3,3,3], index=['x','y','s'])

print('Original Data:\n',df1,'\n\n',
      'Add By Rows: \n', df1.add(s3, axis=0), '\n\n',
      'Add By Columns: \n', df1.add(s4, axis=1))
```

```
#:> Original Data:
#:>         x  y    z
#:> idx
#:> row1  10  1  0.1
#:> row2  20  2  0.2
#:> row3  30  3  0.3
#:>
#:>  Add By Rows:
#:>           x    y    z
#:> row1  11.0  2.0  1.1
#:> row2  21.0  3.0  1.2
```

```
#:> row3   NaN  NaN  NaN
#:> row4   NaN  NaN  NaN
#:>
#:>  Add By Columns:
#:>         s     x    y   z
#:> idx
#:> row1 NaN  13.0  4.0 NaN
#:> row2 NaN  23.0  5.0 NaN
#:> row3 NaN  33.0  6.0 NaN
```

13.7.2.3 Substraction

```
r1 = df2 - df1
r2 = df2.sub(df1,fill_value=1000)

print( r1, '\n\n', r2)
```

```
#:>        k    x   y    z
#:> idx
#:> row1 NaN  3.0 NaN  0.0
#:> row2 NaN  3.0 NaN  0.0
#:> row3 NaN  NaN NaN  NaN
#:> row4 NaN  NaN NaN  NaN
#:>
#:>            k      x      y      z
#:> idx
#:> row1 -989.0    3.0  999.0    0.0
#:> row2 -979.0    3.0  998.0    0.0
#:> row3    NaN  970.0  997.0  999.7
#:> row4 -969.0 -967.0    NaN -999.7
```

```
r3 = (r2>0) & (r2<=3)
print( 'Original Data: \n', r2, '\n\n',
       'Logical Operator:\n', r3)
```

```
#:> Original Data:
#:>            k      x      y      z
#:> idx
#:> row1 -989.0    3.0  999.0    0.0
#:> row2 -979.0    3.0  998.0    0.0
#:> row3    NaN  970.0  997.0  999.7
#:> row4 -969.0 -967.0    NaN -999.7
#:>
#:>  Logical Operator:
#:>            k      x      y      z
#:> idx
```

```
#:> row1  False   True  False  False
#:> row2  False   True  False  False
#:> row3  False  False  False  False
#:> row4  False  False  False  False
```

13.7.3 Attributes

```
df = pd.DataFrame(
    { 'empID':  [100,      101,    102,      103,     104],
      'year1':   [2017,     2017,   2017,      2018,    2018],
      'name':   ['Alice',  'Bob',  'Charles','David', 'Eric'],
      'year2':   [2001,     1907,   2003,      1998,    2011],
      'salary': [40000,    24000,  31000,     20000,   30000]},
    columns = ['year1','salary','year2','empID','name'])
```

13.7.3.1 Dimensions

```
df.shape
```

```
#:> (5, 5)
```

13.7.3.2 Index

```
df.index
```

```
#:> RangeIndex(start=0, stop=5, step=1)
```

Underlying Index values are numpy object

```
df.index.values
```

```
#:> array([0, 1, 2, 3, 4])
```

13.7.3.3 Columns

```
df.columns
```

```
#:> Index(['year1', 'salary', 'year2', 'empID', 'name'], dtype='object')
```

Underlying Index values are numpy object

```
df.columns.values
```

```
#:> array(['year1', 'salary', 'year2', 'empID', 'name'], dtype=object)
```

13.7.3.4 Values

Underlying Column values are numpy object

```
df.values
```

```
#:> array([[2017, 40000, 2001, 100, 'Alice'],
#:>        [2017, 24000, 1907, 101, 'Bob'],
#:>        [2017, 31000, 2003, 102, 'Charles'],
#:>        [2018, 20000, 1998, 103, 'David'],
#:>        [2018, 30000, 2011, 104, 'Eric']], dtype=object)
```

13.7.4 Index Manipulation

index and **row label** are used interchangeably in this book

13.7.4.1 Sample Data

Columns are intentionaly ordered in a messy way

```
df = pd.DataFrame(
    { 'empID':  [100,      101,    102,      103,     104],
      'year1':   [2017,     2017,   2017,      2018,    2018],
      'name':   ['Alice',  'Bob',  'Charles','David', 'Eric'],
      'year2':   [2001,     1907,   2003,      1998,    2011],
      'salary': [40000,    24000,  31000,     20000,   30000]},
    columns = ['year1','salary','year2','empID','name'])

print (df, '\n')
```

```
#:>    year1  salary  year2  empID     name
#:> 0   2017   40000   2001    100    Alice
#:> 1   2017   24000   1907    101      Bob
#:> 2   2017   31000   2003    102  Charles
#:> 3   2018   20000   1998    103    David
#:> 4   2018   30000   2011    104     Eric
```

```
print (df.index)
```

```
#:> RangeIndex(start=0, stop=5, step=1)
```

13.7.4.2 Convert Column To Index

`set_index('column_name', inplace=False)`

inplace=True means don't create a new dataframe. Modify existing dataframe
inplace=False means return a new dataframe

```
print(df)
```

```
#:>    year1  salary  year2  empID     name
#:> 0   2017   40000   2001    100    Alice
#:> 1   2017   24000   1907    101      Bob
```

```
#:> 2   2017   31000   2003    102  Charles
#:> 3   2018   20000   1998    103    David
#:> 4   2018   30000   2011    104     Eric
```

```
print(df.index,'\n')
```

```
#:> RangeIndex(start=0, stop=5, step=1)
```

```
df.set_index('empID',inplace=True)
print(df)
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

```
print(df.index) # return new DataFrameObj
```

```
#:> Int64Index([100, 101, 102, 103, 104], dtype='int64', name='empID')
```

13.7.4.3 Convert Index Back To Column

- Reseting index will resequence the index as 0,1,2 etc
- Old index column will be converted back as normal column
- Operation support inplace** option

```
df.reset_index(inplace=True)
print(df)
```

```
#:>    empID  year1  salary  year2     name
#:> 0    100   2017   40000   2001    Alice
#:> 1    101   2017   24000   1907      Bob
#:> 2    102   2017   31000   2003  Charles
#:> 3    103   2018   20000   1998    David
#:> 4    104   2018   30000   2011     Eric
```

13.7.4.4 Updating Index (.index=)

Warning:
- Updating index **doesn't reorder** the data sequence
- Number of elements before and after reorder must match, otherwise **error**
- Same label are **allowed to repeat** - Not reversable

```
df.index = [101, 101, 101, 102, 103]
df
```

```
#:>      empID  year1  salary  year2     name
#:> 101    100   2017   40000   2001    Alice
#:> 101    101   2017   24000   1907      Bob
#:> 101    102   2017   31000   2003  Charles
#:> 102    103   2018   20000   1998    David
#:> 103    104   2018   30000   2011     Eric
```

13.7.4.5 Reordering Index (. reindex)

- Reindex will **reorder** the rows according to new index
- The operation is not reversable

Start from this original dataframe

```
df.index = [101,102,103,104,105]
df
```

```
#:>      empID  year1  salary  year2     name
#:> 101    100   2017   40000   2001    Alice
#:> 102    101   2017   24000   1907      Bob
#:> 103    102   2017   31000   2003  Charles
#:> 104    103   2018   20000   1998    David
#:> 105    104   2018   30000   2011     Eric
```

Change the order of Index, always return a new dataframe

```
df.reindex([103,102,101,104,105])
```

```
#:>      empID  year1  salary  year2     name
#:> 103    102   2017   31000   2003  Charles
#:> 102    101   2017   24000   1907      Bob
#:> 101    100   2017   40000   2001    Alice
#:> 104    103   2018   20000   1998    David
#:> 105    104   2018   30000   2011     Eric
```

13.7.4.6 Rename Index

- Example below renamed the axis of both columns and rows
- Use `axis=0` for row index, use `axis=1` for column index

```
df.rename_axis('super_id').rename_axis('my_cols', axis=1)
```

```
#:> my_cols   empID  year1  salary  year2     name
#:> super_id
#:> 101         100   2017   40000   2001    Alice
#:> 102         101   2017   24000   1907      Bob
#:> 103         102   2017   31000   2003  Charles
#:> 104         103   2018   20000   1998    David
```

```
#:> 105         104   2018   30000   2011     Eric
```

13.7.5 Subsetting Columns

Select Single Column Return **Series**

```
dataframe.columnName               # single column, name based, return Series object
dataframe[ single_col_name ]       # single column, name based, return Series object
dataframe[ [single_col_name] ]     # single column, name based, return DataFrame objec
```

Select Single/Multiple Columns Return **DataFrame**

```
dataframe[ single/list_of_col_names ]                       # name based, return Dataf
dataframe.loc[ : , single_col_name  ]  # single column, series
dataframe.loc[ : , col_name_list    ]  # multiple columns, dataframe
dataframe.loc[ : , col_name_ranage  ]  # multiple columns, dataframe

dataframe.iloc[ : , col_number      ]  # single column, series
dataframe.iloc[ : , col_number_list ]  # multiple columns, dataframe
dataframe.iloc[ : , number_range    ]  # multiple columns, dataframe
```

13.7.5.1 Select Single Column

Selecting single column always return as **panda::Series**

```
print(
  df.name,           '\n\n',
  df['name'],        '\n\n',
  df.loc[:, 'name'], '\n\n',
  df.iloc[:, 3])
```

```
#:> 101      Alice
#:> 102        Bob
#:> 103    Charles
#:> 104      David
#:> 105       Eric
#:> Name: name, dtype: object
#:>
#:>  101      Alice
#:> 102        Bob
#:> 103    Charles
#:> 104      David
#:> 105       Eric
#:> Name: name, dtype: object
#:>
#:>  101      Alice
#:> 102        Bob
#:> 103    Charles
```

```
#:> 104      David
#:> 105       Eric
#:> Name: name, dtype: object
#:>
#:>  101    2001
#:> 102    1907
#:> 103    2003
#:> 104    1998
#:> 105    2011
#:> Name: year2, dtype: int64
```

13.7.5.2 Select Multiple Columns

Multiple columns return as **panda::Dataframe** object‘

Example below returns DataFrame with Single Column

```
df[['name']]  # return one column dataframe
```

```
#:>        name
#:> 101    Alice
#:> 102      Bob
#:> 103  Charles
#:> 104    David
#:> 105     Eric
```

```
print(
  df[['name','year1']]       ,'\n\n',
  df.loc[:,['name','year1']])
```

```
#:>        name  year1
#:> 101    Alice   2017
#:> 102      Bob   2017
#:> 103  Charles   2017
#:> 104    David   2018
#:> 105     Eric   2018
#:>
#:>         name  year1
#:> 101    Alice   2017
#:> 102      Bob   2017
#:> 103  Charles   2017
#:> 104    David   2018
#:> 105     Eric   2018
```

Select Range of Columns

```
print(
  df.loc [ : , 'year1':'year2'], '\n\n',
  df.iloc[ : , [0,3]]           ,'\n\n',
```

```
  df.iloc[ : , 0:3]
)
```

```
#:>      year1  salary  year2
#:> 101   2017   40000   2001
#:> 102   2017   24000   1907
#:> 103   2017   31000   2003
#:> 104   2018   20000   1998
#:> 105   2018   30000   2011
#:>
#:>       empID  year2
#:> 101    100   2001
#:> 102    101   1907
#:> 103    102   2003
#:> 104    103   1998
#:> 105    104   2011
#:>
#:>       empID  year1  salary
#:> 101    100   2017   40000
#:> 102    101   2017   24000
#:> 103    102   2017   31000
#:> 104    103   2018   20000
#:> 105    104   2018   30000
```

13.7.5.3 By Column Name (.filter)

`.filter(items=None, like=None, regex=None, axis=1)`

like = Substring Matches

```
df.filter( like='year',  axis='columns')  ## or axis = 1
```

```
#:>      year1  year2
#:> 101   2017   2001
#:> 102   2017   1907
#:> 103   2017   2003
#:> 104   2018   1998
#:> 105   2018   2011
```

items = list of column names

```
df.filter( items=('year1','year2'),  axis=1)  ## or axis = 1
```

```
#:>      year1  year2
#:> 101   2017   2001
#:> 102   2017   1907
#:> 103   2017   2003
#:> 104   2018   1998
```

```
#:> 105   2018   2011
```

regex = Regular Expression
Select column names that contain integer

```
df.filter(regex='\d')  ## default axis=1 if DataFrame
```

```
#:>      year1  year2
#:> 101   2017   2001
#:> 102   2017   1907
#:> 103   2017   2003
#:> 104   2018   1998
#:> 105   2018   2011
```

13.7.5.4 Data Type (.select_dtypes)

```
df.select_dtypes(include=None, exclude=None)
```

Always return **panda::DataFrame**, even though only single column matches.
Allowed types are: - number (integer and float)
- integer / float - datetime
- timedelta
- category

```
# error: no attribute get_dtype_counts
df.get_dtype_counts()
```

```
df.select_dtypes(exclude='number')
```

```
#:>         name
#:> 101    Alice
#:> 102      Bob
#:> 103  Charles
#:> 104    David
#:> 105     Eric
```

```
df.select_dtypes(exclude=('number','object'))
```

```
#:> Empty DataFrame
#:> Columns: []
#:> Index: [101, 102, 103, 104, 105]
```

13.7.6 Column Manipulation

13.7.6.1 Sample Data

```
df
```

```
#:>      empID  year1  salary  year2     name
#:> 101    100   2017   40000   2001    Alice
```

```
#:> 102    101   2017   24000   1907      Bob
#:> 103    102   2017   31000   2003  Charles
#:> 104    103   2018   20000   1998    David
#:> 105    104   2018   30000   2011     Eric
```

13.7.6.2 Renaming Columns

Method 1 : Rename All Columns (.columns =)
- Construct the new column names, **check if there is no missing** column names
- **Missing columns** will return **error**
- Direct Assignment to column property result in change to dataframe

```
new_columns = ['year.1','salary','year.2','empID','name']
df.columns = new_columns
df.head(2)
```

```
#:>      year.1  salary  year.2  empID   name
#:> 101     100    2017   40000   2001  Alice
#:> 102     101    2017   24000   1907    Bob
```

Method 2 : Renaming Specific Column (.rename (columns=)) - Change column name through **rename** function
- Support **inpalce** option for original dataframe change
- Missing column is OK

```
df.rename( columns={'year.1':'year1', 'year.2':'year2'}, inplace=True)
df.head(2)
```

```
#:>      year1  salary  year2  empID   name
#:> 101    100    2017  40000   2001  Alice
#:> 102    101    2017  24000   1907    Bob
```

13.7.6.3 Reordering Columns

Always return a new dataframe. There is **no inplace option** for reordering columns

Method 1 - reindex(columns =)
- **reindex** may sounds like operation on row labels, but it works
- **Missmatch** column names will result in **NA** for the unfound column

```
new_colorder = [ 'empID', 'name', 'salary', 'year1', 'year2']
df.reindex(columns = new_colorder).head(2)
```

```
#:>      empID   name  salary  year1  year2
#:> 101   2001  Alice    2017    100  40000
#:> 102   1907    Bob    2017    101  24000
```

Method 2 - [ ] notation
- **Missmatch** column will result in **ERROR**

```
new_colorder = [ 'empID', 'name', 'salary', 'year1', 'year2']
df[new_colorder]
```

```
#:>      empID     name  salary  year1  year2
#:> 101   2001    Alice    2017    100  40000
#:> 102   1907      Bob    2017    101  24000
#:> 103   2003  Charles    2017    102  31000
#:> 104   1998    David    2018    103  20000
#:> 105   2011     Eric    2018    104  30000
```

13.7.6.4 Duplicating or Replacing Column

- **New Column** will be created instantly using [] **notation**
- **DO NOT USE dot Notation** because it is view only attribute

```
df['year3'] = df.year1
df
```

```
#:>      year1  salary  year2  empID     name  year3
#:> 101    100    2017  40000   2001    Alice    100
#:> 102    101    2017  24000   1907      Bob    101
#:> 103    102    2017  31000   2003  Charles    102
#:> 104    103    2018  20000   1998    David    103
#:> 105    104    2018  30000   2011     Eric    104
```

13.7.6.5 Dropping Columns (.drop)

```
dataframe.drop( columns='column_name',    inplace=True/False)   # delete single column
dataframe.drop( columns=list_of_colnames, inplace=True/False)   # delete multiple column

dataframe.drop( index='row_label',         inplace=True/False)   # delete single row
dataframe.drop( index= list_of_row_labels, inplace=True/False)   # delete multiple rows
```

inplace=True means column will be deleted from original dataframe. **Default is False**, which return a copy of dataframe

By Column Name(s)

```
df.drop( columns='year1') # drop single column
```

```
#:>      salary  year2  empID     name  year3
#:> 101    2017  40000   2001    Alice    100
#:> 102    2017  24000   1907      Bob    101
#:> 103    2017  31000   2003  Charles    102
#:> 104    2018  20000   1998    David    103
#:> 105    2018  30000   2011     Eric    104
```

```
df.drop(columns=['year2','year3'])  # drop multiple columns
```

```
#:>      year1  salary  empID     name
#:> 101    100    2017   2001    Alice
#:> 102    101    2017   1907      Bob
#:> 103    102    2017   2003  Charles
#:> 104    103    2018   1998    David
#:> 105    104    2018   2011     Eric
```

By Column Number(s)
Use dataframe.columns to produce interim list of column names

```
df.drop( columns=df.columns[[3,4,5]] )   # delete columns by list of column number
```

```
#:>      year1  salary  year2
#:> 101    100    2017  40000
#:> 102    101    2017  24000
#:> 103    102    2017  31000
#:> 104    103    2018  20000
#:> 105    104    2018  30000
```

```
df.drop( columns=df.columns[3:6] )       # delete columns by range of column number
```

```
#:>      year1  salary  year2
#:> 101    100    2017  40000
#:> 102    101    2017  24000
#:> 103    102    2017  31000
#:> 104    103    2018  20000
#:> 105    104    2018  30000
```

13.7.7 Subsetting Rows

```
dataframe.loc[ row_label       ]  # return series, single row
dataframe.loc[ row_label_list  ]  # multiple rows
dataframe.loc[ boolean_list    ]  # multiple rows

dataframe.iloc[ row_number      ]  # return series, single row
dataframe.iloc[ row_number_list  ]  # multiple rows
dataframe.iloc[ number_range     ]  # multiple rows

dataframe.sample(frac=)                                # frac = 0.6 means sampl
```

13.7.7.1 Sample Data

```
df = pd.DataFrame(
    { 'empID':  [100,      101,    102,      103,     104],
      'year1':   [2017,     2017,   2017,      2018,    2018],
```

```
      'name':   ['Alice',  'Bob',  'Charles','David', 'Eric'],
      'year2':   [2001,     1907,   2003,      1998,    2011],
      'salary': [40000,    24000,  31000,     20000,   30000]},
    columns = ['year1','salary','year2','empID','name']).set_index(['empID'])
df
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

13.7.7.2 By Index or Boolean

Single Index return Series

```
df.loc[101]         # by single row label, return series
```

```
#:> year1      2017
#:> salary    24000
#:> year2      1907
#:> name        Bob
#:> Name: 101, dtype: object
```

List or Range of Indexes returns DataFrame

```
df.loc[ [100,103] ]  # by multiple row labels
```

```
#:>        year1  salary  year2   name
#:> empID
#:> 100     2017   40000   2001  Alice
#:> 103     2018   20000   1998  David
```

```
df.loc[  100:103  ]  # by range of row labels
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
```

List of Boolean returns DataFrame

```
criteria = (df.salary > 30000) & (df.year1==2017)
print (criteria)
```

```
#:> empID
```

```
#:> 100     True
#:> 101    False
#:> 102     True
#:> 103    False
#:> 104    False
#:> dtype: bool
```

```
print (df.loc[criteria])
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 102     2017   31000   2003  Charles
```

13.7.7.3 By Row Number

Single Row return Series

```
df.iloc[1]  # by single row number
```

```
#:> year1      2017
#:> salary    24000
#:> year2      1907
#:> name        Bob
#:> Name: 101, dtype: object
```

Multiple rows **returned as dataframe** object

```
df.iloc[ [0,3] ]    # by row numbers
```

```
#:>        year1  salary  year2   name
#:> empID
#:> 100     2017   40000   2001  Alice
#:> 103     2018   20000   1998  David
```

```
df.iloc[  0:3  ]    # by row number range
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
```

13.7.7.4 By Expression (.query)

`.query(expr, inplace=False)`

```
df.query('salary<=31000 and year1 == 2017')
```

```
#:>        year1  salary  year2     name
```

```
#:> empID
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
```

13.7.7.5 By Random (.sample)

```
np.random.seed(15)
df.sample(frac=0.6) #randomly pick 60% of rows, without replacement
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

13.7.8 Row Manipulation

13.7.8.1 Sample Data

13.7.8.2 Appending Rows

Appending rows is more computaional intensive then concatenate. Item can be added as single item or multi-items (list form)

Append From Another DataFrame

- When `ignore_index=True`, pandas will **drop the original Index** and recreate with 0,1,2,3…
- It is recommended to ignore index IF the data source index is **not unique**.
- New columns will be added in the result, with NaN on original dataframe.

```
my_df = pd.DataFrame(
          data= {'Id':   [10,20,30],
                 'Name': ['Aaa','Bbb','Ccc']})
#                 .set_index('Id')

my_df_new = pd.DataFrame(
            data= {'Id':   [40,50],
                   'Name': ['Ddd','Eee'],
                   'Age':  [12,13]})
                   #.set_index('Id')

my_df_append  = my_df.append(my_df_new, ignore_index=False)
my_df_noindex = my_df.append(my_df_new, ignore_index=True)

print("Original DataFrame:\n", my_df,
```

```
      "\n\nTo Be Appended DataFrame:\n", my_df_new,
      "\n\nAppended DataFrame (index maintained):\n", my_df_append,
      "\n\nAppended DataFrame (index ignored):\n", my_df_noindex)
```

```
#:> Original DataFrame:
#:>     Id Name
#:> 0  10  Aaa
#:> 1  20  Bbb
#:> 2  30  Ccc
#:>
#:> To Be Appended DataFrame:
#:>     Id Name  Age
#:> 0  40  Ddd   12
#:> 1  50  Eee   13
#:>
#:> Appended DataFrame (index maintained):
#:>     Id Name   Age
#:> 0  10  Aaa   NaN
#:> 1  20  Bbb   NaN
#:> 2  30  Ccc   NaN
#:> 0  40  Ddd  12.0
#:> 1  50  Eee  13.0
#:>
#:> Appended DataFrame (index ignored):
#:>     Id Name   Age
#:> 0  10  Aaa   NaN
#:> 1  20  Bbb   NaN
#:> 2  30  Ccc   NaN
#:> 3  40  Ddd  12.0
#:> 4  50  Eee  13.0
```

Append From Dictionary

```
my_df = pd.DataFrame(
          data= {'Id':   [10,20,30],
                 'Name': ['Aaa','Bbb','Ccc']})  \
                 .set_index('Id')

new_item1 = {'Id':40, 'Name': 'Ddd'}
new_item2 = {'Id':50, 'Name': 'Eee'}
new_item3 = {'Id':60, 'Name': 'Fff'}

my_df_one   = my_df.append( new_item1, ignore_index=True )
my_df_multi = my_df.append( [new_item2, new_item3], ignore_index=True )

print("Original DataFrame:\n", my_df,
```

```
      "\n\nAdd One Item (index ignored):\n", my_df_one,
      "\n\nAdd Multi Item (index ignored):\n", my_df_multi)
```

```
#:> Original DataFrame:
#:>     Name
#:> Id
#:> 10  Aaa
#:> 20  Bbb
#:> 30  Ccc
#:>
#:> Add One Item (index ignored):
#:>    Name    Id
#:> 0  Aaa   NaN
#:> 1  Bbb   NaN
#:> 2  Ccc   NaN
#:> 3  Ddd  40.0
#:>
#:> Add Multi Item (index ignored):
#:>    Name    Id
#:> 0  Aaa   NaN
#:> 1  Bbb   NaN
#:> 2  Ccc   NaN
#:> 3  Eee  50.0
#:> 4  Fff  60.0
```

Appending `None` items(s)

Adding **single None** item has **no effect** (nothing added).
Adding **None in list form (multiple items)** creates rows with None.
`ignore_index` is not important here.

```
single_none = my_df.append( None  )
multi_none  = my_df.append( [None])

print("Original DataFrame:\n", my_df,
      "\n\nAdd One None (index ignored):\n", single_none,
      "\n\nAdd List of None (index ignored):\n", multi_none)
```

```
#:> Original DataFrame:
#:>     Name
#:> Id
#:> 10  Aaa
#:> 20  Bbb
#:> 30  Ccc
#:>
#:> Add One None (index ignored):
#:>     Name
```

```
#:> Id
#:> 10  Aaa
#:> 20  Bbb
#:> 30  Ccc
#:>
#:> Add List of None (index ignored):
#:>     Name     0
#:> 10  Aaa   NaN
#:> 20  Bbb   NaN
#:> 30  Ccc   NaN
#:> 0   NaN  None
```

Appending Items Containing None results in **ERROR**

```
# error
my_df.append( [new_item1, None] )
```

13.7.8.3 Concatenate Rows

13.7.8.4 Dropping Rows (.drop)

`.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')`

By Row Label(s)

```
df.drop(index=100)       # single row
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

```
df.drop(index=[100,103])   # multiple rows
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 104     2018   30000   2011     Eric
```

13.7.9 Slicing

13.7.9.1 Sample Data

```
df
```

```
#:>        year1  salary  year2     name
```

```
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

13.7.9.2 Getting One Cell

By Row Label and Column Name (loc)

```
dataframe.loc [ row_label , col_name   ]    # by row label and column names
dataframe.loc [ bool_list , col_name   ]    # by row label and column names
dataframe.iloc[ row_number, col_number ]    # by row and column number
```

```
print (df.loc[100,'year1'])
```

```
#:> 2017
```

By Row Number and Column Number (iloc)

```
print (df.iloc[1,2])
```

```
#:> 1907
```

13.7.9.3 Getting Multiple Cells

Specify rows and columns (by individual or range)

```
dataframe.loc [ list/range_of_row_labels , list/range_col_names   ]    # by row label and column
dataframe.iloc[ list/range_row_numbers,    list/range_col_numbers ]    # by row number
```

By Index and Column Name (loc)

```
print (df.loc[ [101,103], ['name','year1'] ], '\n')  # by list of row label and column names
```

```
#:>         name  year1
#:> empID
#:> 101      Bob   2017
#:> 103    David   2018
```

```
print (df.loc[  101:104 ,  'year1':'year2'  ], '\n')  # by range of row label and column names
```

```
#:>        year1  salary  year2
#:> empID
#:> 101     2017   24000   1907
#:> 102     2017   31000   2003
#:> 103     2018   20000   1998
#:> 104     2018   30000   2011
```

By Boolean Row and Column Names (loc)

```
df.loc[df.year1==2017, 'year1':'year2']
```

```
#:>        year1  salary  year2
#:> empID
#:> 100     2017   40000   2001
#:> 101     2017   24000   1907
#:> 102     2017   31000   2003
```

By Row and Column Number (iloc)

```
print (df.iloc[ [1,4], [0,3]],'\n' )   # by individual rows/columns
```

```
#:>        year1  name
#:> empID
#:> 101     2017   Bob
#:> 104     2018  Eric
```

```
print (df.iloc[  1:4 ,  0:3], '\n')    # by range
```

```
#:>        year1  salary  year2
#:> empID
#:> 101     2017   24000   1907
#:> 102     2017   31000   2003
#:> 103     2018   20000   1998
```

13.7.10 Chained Indexing

Chained Index Method creates a copy of dataframe, any modification of data on original dataframe does not affect the copy

```
dataframe.loc  [...]  [...]
dataframe.iloc [...]  [...]
```

Suggesting, **never use** chain indexing

```
df = pd.DataFrame(
    { 'empID':  [100,      101,    102,      103,     104],
      'year1':   [2017,     2017,   2017,      2018,    2018],
      'name':   ['Alice',  'Bob',  'Charles','David', 'Eric'],
      'year2':   [2001,     1907,   2003,      1998,    2011],
      'salary': [40000,    24000,  31000,     20000,   30000]},
    columns = ['year1','salary','year2','empID','name']).set_index(['empID'])
df
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
```

```
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

```
df.loc[100]['year'] =2000
```

```
#:> /home/msfz751/miniconda3/envs/python_book/bin/python:1: SettingWithCopyWarning:
#:> A value is trying to be set on a copy of a slice from a DataFrame
#:>
#:> See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-
copy
#:> /home/msfz751/miniconda3/envs/python_book/lib/python3.7/site-
packages/pandas/core/indexing.py:670: SettingWithCopyWarning:
#:> A value is trying to be set on a copy of a slice from a DataFrame
#:>
#:> See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-
copy
#:>   iloc._setitem_with_indexer(indexer, value)
```

```
df  ## notice row label 100 had not been updated, because data was updated on a copy due to chai
```

```
#:>        year1  salary  year2     name
#:> empID
#:> 100     2017   40000   2001    Alice
#:> 101     2017   24000   1907      Bob
#:> 102     2017   31000   2003  Charles
#:> 103     2018   20000   1998    David
#:> 104     2018   30000   2011     Eric
```

13.7.11 Cell Value Replacement

Slicing deals with square cells selection. Use `mask` or `where` to select specific cell(s). These function respect column and row names.

13.7.11.1 `mask()`

`mask()` replace value with `other=` when condition is met. Column and row name is **respected**

```
ori = pd.DataFrame(data={
     'x': [1,4,7],
     'y': [2,5,8],
     'z': [3,6,9]}, index=[
     'row1','row2','row3'])

df_big = (ori >4)[['y','x','z']]
resul1 = ori.mask(df_big, other=999)
```

```
print('Original DF: \n', ori, '\n\n',
      'Big DF : \n', df_big, '\n\n',
      'Result : \n', resul1)
```

```
#:> Original DF:
#:>        x  y  z
#:> row1  1  2  3
#:> row2  4  5  6
#:> row3  7  8  9
#:>
#:>  Big DF :
#:>            y      x      z
#:> row1  False  False  False
#:> row2   True  False   True
#:> row3   True   True   True
#:>
#:>  Result :
#:>          x    y    z
#:> row1    1    2    3
#:> row2    4  999  999
#:> row3  999  999  999
```

13.7.11.2 `where()`

This is reverse of `mask()`, it will repalce value when the **condition is False**.

```
df.where(cond=df_big)
```

```
#:>        year1  salary  year2 name
#:> empID
#:> 100      NaN     NaN    NaN  NaN
#:> 101      NaN     NaN    NaN  NaN
#:> 102      NaN     NaN    NaN  NaN
#:> 103      NaN     NaN    NaN  NaN
#:> 104      NaN     NaN    NaN  NaN
```

13.7.12 Iteration

13.7.12.1 Loop Through Rows (.iterrows)

```
df = pd.DataFrame(data=
    { 'empID':  [100,      101,    102,      103,     104],
      'Name':   ['Alice',  'Bob',  'Charles','David', 'Eric'],
      'Year':   [1999,     1988,   2001,     2010,     2020]}).set_index(['empID'])
```

```
for idx, row in df.iterrows():
  print(idx, row.Name)
```

```
#:> 100 Alice
#:> 101 Bob
#:> 102 Charles
#:> 103 David
#:> 104 Eric
```

13.7.12.2 Loop Through Columns (.itemes)

```
for label, content in df.items():
  print('Label:',            label,   '\n\n',
        'Content (Series):\n', content, '\n\n')
```

```
#:> Label: Name
#:>
#:>  Content (Series):
#:>  empID
#:> 100      Alice
#:> 101        Bob
#:> 102    Charles
#:> 103      David
#:> 104       Eric
#:> Name: Name, dtype: object
#:>
#:>
#:> Label: Year
#:>
#:>  Content (Series):
#:>  empID
#:> 100    1999
#:> 101    1988
#:> 102    2001
#:> 103    2010
#:> 104    2020
#:> Name: Year, dtype: int64
```

13.7.13 Data Structure

13.7.13.1 Instance Methods - Structure

Find out the column names, data type in a summary. Output is for display only, not a data object

```
df.info()  # return text output
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> Int64Index: 5 entries, 100 to 104
#:> Data columns (total 2 columns):
#:>  #   Column  Non-Null Count  Dtype
#:> ---  ------  --------------  -----
#:>  0   Name    5 non-null      object
#:>  1   Year    5 non-null      int64
#:> dtypes: int64(1), object(1)
#:> memory usage: 120.0+ bytes
```

```
df.get_dtype_counts() # return Series
```

13.7.13.2 Conversion To Other Format

```
df.to_json()
```

```
#:> '{"Name":{"100":"Alice","101":"Bob","102":"Charles","103":"David","104":"Eric"},"Ye
```

```
df.to_records()
```

```
#:> rec.array([(100, 'Alice', 1999), (101, 'Bob', 1988),
#:>            (102, 'Charles', 2001), (103, 'David', 2010),
#:>            (104, 'Eric', 2020)],
#:>           dtype=[('empID', '<i8'), ('Name', 'O'), ('Year', '<i8')])
```

```
df.to_csv()
```

```
#:> 'empID,Name,Year\n100,Alice,1999\n101,Bob,1988\n102,Charles,2001\n103,David,2010\n1
```

13.8 class: MultiIndex

MultiIndexing are columns with few levels of headers.

13.8.1 The Data

```
df = pd.DataFrame({
     'myindex': [0, 1, 2],
     'One_X':   [1.1,  1.1,  1.1],
     'One_Y':   [1.2,  1.2,  1.2],
     'Two_X':   [1.11, 1.11, 1.11],
     'Two_Y':   [1.22, 1.22, 1.22]})
df.set_index('myindex',inplace=True)
df
```

```
#:>          One_X  One_Y  Two_X  Two_Y
```

```
#:> myindex
#:> 0          1.1    1.2   1.11   1.22
#:> 1          1.1    1.2   1.11   1.22
#:> 2          1.1    1.2   1.11   1.22
```

13.8.2 Creating MultiIndex Object

13.8.2.1 Create From Tuples

MultiIndex can easily created from typles:
- Step 1: Create a MultiIndex object by splitting column name into tuples
- Step 2: Assign the MultiIndex Object to dataframe `columns` property.

```
my_tuples = [tuple(c.split('_')) for c in df.columns]
df.columns = pd.MultiIndex.from_tuples(my_tuples)

print(' Column Headers :\n\n',           my_tuples,
        '\n\nNew Columns: \n\n',         df.columns,
        '\n\nTwo Layers Header DF:\n\n', df)
```

```
#:>  Column Headers :
#:>
#:>  [('One', 'X'), ('One', 'Y'), ('Two', 'X'), ('Two', 'Y')]
#:>
#:> New Columns:
#:>
#:>  MultiIndex([('One', 'X'),
#:>             ('One', 'Y'),
#:>             ('Two', 'X'),
#:>             ('Two', 'Y')],
#:>            )
#:>
#:> Two Layers Header DF:
#:>
#:>          One        Two
#:>            X    Y     X     Y
#:> myindex
#:> 0        1.1  1.2  1.11  1.22
#:> 1        1.1  1.2  1.11  1.22
#:> 2        1.1  1.2  1.11  1.22
```

13.8.3 MultiIndex Object

13.8.3.1 Levels

- MultiIndex object contain multiple leveels, each level (header) is an Index object.

- Use **`MultiIndex.get_level_values()`** to the entire header for the desired level. Note that each level is an Index object

```
print(df.columns.get_level_values(0), '\n',
      df.columns.get_level_values(1))
```

```
#:> Index(['One', 'One', 'Two', 'Two'], dtype='object')
#:>  Index(['X', 'Y', 'X', 'Y'], dtype='object')
```

`MultiIndex.levels` return the **unique values** of each level.

```
print(df.columns.levels[0], '\n',
      df.columns.levels[1])
```

```
#:> Index(['One', 'Two'], dtype='object')
#:>  Index(['X', 'Y'], dtype='object')
```

13.8.3.2 Convert MultiIndex Back To Tuples

```
df.columns.to_list()
```

```
#:> [('One', 'X'), ('One', 'Y'), ('Two', 'X'), ('Two', 'Y')]
```

13.8.4 Selecting Column(s)

13.8.4.1 Sample Data

```
import itertools
test_df = pd.DataFrame
max_age = 100

### Create The Columns Tuple
level0_sex = ['Male','Female','Pondan']
level1_age = ['Medium','High','Low']
my_columns = list(itertools.product(level0_sex, level1_age))

test_df = pd.DataFrame([
             [1,2,3,4,5,6,7,8,9],
             [11,12,13,14,15,16,17,18,19],
             [21,22,23,24,25,26,27,28,29]], index=['row1','row2','row3'])

### Create Multiindex From Tuple
test_df.columns = pd.MultiIndex.from_tuples(my_columns)
print( test_df )
```

```
#:>        Male            Female            Pondan
#:>      Medium High Low Medium High Low Medium High Low
#:> row1      1    2   3      4    5   6      7    8   9
```

```
#:> row2     11   12  13     14   15  16     17   18  19
#:> row3     21   22  23     24   25  26     27   28  29
```

13.8.4.2 Select Level0 Header(s)

Use **[L0] notation**, where L0 is list of header names

```
print( test_df[['Male','Pondan']] ,'\n\n',  ## Include multiple Level0 Header
       test_df['Male'] ,          '\n\n',   ## Include single Level0 Header
       test_df.Male )                       ## Same as above
```

```
#:>        Male          Pondan
#:>      Medium High Low Medium High Low
#:> row1      1    2   3      7    8   9
#:> row2     11   12  13     17   18  19
#:> row3     21   22  23     27   28  29
#:>
#:>        Medium  High  Low
#:> row1       1     2    3
#:> row2      11    12   13
#:> row3      21    22   23
#:>
#:>        Medium  High  Low
#:> row1       1     2    3
#:> row2      11    12   13
#:> row3      21    22   23
```

Using .loc[]

Use **.loc[:, L0]**, where L0 is list of headers names

```
print( test_df.loc[:, ['Male','Pondan']] , '\n\n',  ## Multiple Level0 Header
       test_df.loc[:, 'Male'] )                     ## Single Level0 Header
```

```
#:>        Male          Pondan
#:>      Medium High Low Medium High Low
#:> row1      1    2   3      7    8   9
#:> row2     11   12  13     17   18  19
#:> row3     21   22  23     27   28  29
#:>
#:>        Medium  High  Low
#:> row1       1     2    3
#:> row2      11    12   13
#:> row3      21    22   23
```

13.8.4.3 Selecting Level 1 Header(s)

Use **.loc[:, (All, L1)]**, where L1 are list of headers names

```
All = slice(None)
print( test_df.loc[ : , (All, 'High')],  '\n\n',  ## Signle L1 header
       test_df.loc[ : , (All, ['High','Low'])] )  ## Multiple L1 headers
```

```
#:>      Male Female Pondan
#:>      High   High   High
#:> row1    2      5      8
#:> row2   12     15     18
#:> row3   22     25     28
#:>
#:>       Male     Female     Pondan
#:>      High Low   High Low   High Low
#:> row1    2   3      5   6      8   9
#:> row2   12  13     15  16     18  19
#:> row3   22  23     25  26     28  29
```

13.8.4.4 Select Level 0 and Level1 Headers

Use `.loc[ :, (L0, L1)]`, where `L0` and `L1` are list of headers names

```
test_df.loc[ : , (['Male','Pondan'], ['Medium','High'])]
```

```
#:>        Male      Pondan
#:>      Medium High Medium High
#:> row1      1    2      7    8
#:> row2     11   12     17   18
#:> row3     21   22     27   28
```

13.8.4.5 Select single L0,L1 Header

Use `.loc[:, (L0,  L1) ]`, result is a **Series**
Use `.loc[:, (L0 ,[L1])]`, result is a **DataFrame**

```
print( test_df.loc[ : , ('Female', 'High')], '\n\n',
       test_df.loc[ : , ('Female', ['High'])])
```

```
#:> row1     5
#:> row2    15
#:> row3    25
#:> Name: (Female, High), dtype: int64
#:>
#:>       Female
#:>        High
#:> row1      5
#:> row2     15
#:> row3     25
```

13.8.5 Headers Ordering

Note that columns **order** specifeid by `[ ]` selection were not respected. This can be remediated either by Sorting and rearranging.

13.8.5.1 Sort Headers

Use `.sort_index()` on DataFrame to sort the headers. Note that when level1 is sorted, it jumble up level0 headers.

```
test_df_sorted_l0 = test_df.sort_index(axis=1, level=0)
test_df_sorted_l1 = test_df.sort_index(axis=1, level=1, ascending=False)
print(test_df, '\n\n',test_df_sorted_l0, '\n\n', test_df_sorted_l1)
```

```
#:>        Male            Female           Pondan
#:>      Medium High Low Medium High Low Medium High Low
#:> row1      1    2   3      4    5   6      7    8   9
#:> row2     11   12  13     14   15  16     17   18  19
#:> row3     21   22  23     24   25  26     27   28  29
#:>
#:>       Female             Male             Pondan
#:>        High Low Medium High Low Medium   High Low Medium
#:> row1      5   6      4    2   3      1      8   9      7
#:> row2     15  16     14   12  13     11     18  19     17
#:> row3     25  26     24   22  23     21     28  29     27
#:>
#:>       Pondan   Male Female Pondan Male Female Pondan Male Female
#:>      Medium Medium Medium    Low  Low    Low   High High   High
#:> row1      7      1      4      9    3      6      8    2      5
#:> row2     17     11     14     19   13     16     18   12     15
#:> row3     27     21     24     29   23     26     28   22     25
```

13.8.5.2 Rearranging Headers

Use **`.reindex()`** on arrange columns in specific order. Example below shows how to control the specific order for level1 headers.

```
cats = ['Low','Medium','High']
test_df.reindex(cats, level=1, axis=1)
```

```
#:>      Male              Female              Pondan
#:>       Low Medium High    Low Medium High    Low Medium High
#:> row1    3      1    2      6      4    5      9      7    8
#:> row2   13     11   12     16     14   15     19     17   18
#:> row3   23     21   22     26     24   25     29     27   28
```

13.8.6 Stacking and Unstacking

```
df.stack()
```

```
#:>            One   Two
#:> myindex
#:> 0       X  1.1  1.11
#:>         Y  1.2  1.22
#:> 1       X  1.1  1.11
#:>         Y  1.2  1.22
#:> 2       X  1.1  1.11
#:>         Y  1.2  1.22
```

13.8.6.1 Stacking Columns to Rows

Stacking with **`DataFrame.stack(level_no)`** is moving wide columns into row.

```
print('Stacking Header Level 0: \n\n', df.stack(0),
      '\n\nStacking Header Level 1: \n\n', df.stack(1))
```

```
#:> Stacking Header Level 0:
#:>
#:>                  X     Y
#:> myindex
#:> 0       One  1.10  1.20
#:>         Two  1.11  1.22
#:> 1       One  1.10  1.20
#:>         Two  1.11  1.22
#:> 2       One  1.10  1.20
#:>         Two  1.11  1.22
#:>
#:> Stacking Header Level 1:
#:>
#:>             One   Two
#:> myindex
#:> 0       X  1.1  1.11
#:>         Y  1.2  1.22
#:> 1       X  1.1  1.11
#:>         Y  1.2  1.22
#:> 2       X  1.1  1.11
#:>         Y  1.2  1.22
```

13.8.7 Exploratory Analysis

13.8.7.1 Sample Data

```
df
```

```
#:>          One        Two
#:>            X    Y     X     Y
#:> myindex
#:> 0        1.1  1.2  1.11  1.22
#:> 1        1.1  1.2  1.11  1.22
#:> 2        1.1  1.2  1.11  1.22
```

13.8.7.2 All Stats in One - .describe()

```
df.describe(include='number') # default
df.describe(include='object') # display for non-numeric columns
df.describe(include='all')    # display both numeric and non-
numeric
```

When applied to DataFrame object, describe shows all **basic statistic** for **all numeric** columns: - Count (non-NA)
- Unique (for string)
- Top (for string)
- Frequency (for string)
- Percentile
- Mean
- Min / Max
- Standard Deviation

For Numeric Columns only
You can **customize the percentiles requred**. Notice 0.5 percentile is always there although not specified

```
df.describe()
```

```
#:>        One        Two
#:>          X    Y     X     Y
#:> count  3.0  3.0  3.00  3.00
#:> mean   1.1  1.2  1.11  1.22
#:> std    0.0  0.0  0.00  0.00
#:> min    1.1  1.2  1.11  1.22
#:> 25%    1.1  1.2  1.11  1.22
#:> 50%    1.1  1.2  1.11  1.22
#:> 75%    1.1  1.2  1.11  1.22
#:> max    1.1  1.2  1.11  1.22
```

```
df.describe(percentiles=[0.9,0.3,0.2,0.1])
```

```
#:>        One        Two      
#:>          X    Y     X     Y
#:> count  3.0  3.0  3.00  3.00
#:> mean   1.1  1.2  1.11  1.22
#:> std    0.0  0.0  0.00  0.00
#:> min    1.1  1.2  1.11  1.22
#:> 10%    1.1  1.2  1.11  1.22
#:> 20%    1.1  1.2  1.11  1.22
#:> 30%    1.1  1.2  1.11  1.22
#:> 50%    1.1  1.2  1.11  1.22
#:> 90%    1.1  1.2  1.11  1.22
#:> max    1.1  1.2  1.11  1.22
```

For both Numeric and Object

```
df.describe(include='all')
```

```
#:>        One        Two      
#:>          X    Y     X     Y
#:> count  3.0  3.0  3.00  3.00
#:> mean   1.1  1.2  1.11  1.22
#:> std    0.0  0.0  0.00  0.00
#:> min    1.1  1.2  1.11  1.22
#:> 25%    1.1  1.2  1.11  1.22
#:> 50%    1.1  1.2  1.11  1.22
#:> 75%    1.1  1.2  1.11  1.22
#:> max    1.1  1.2  1.11  1.22
```

13.8.7.3 min/max/mean/median

```
df.min()  # default axis=0, column-wise
```

```
#:> One  X    1.10
#:>      Y    1.20
#:> Two  X    1.11
#:>      Y    1.22
#:> dtype: float64
```

```
df.min(axis=1) # axis=1, row-wise
```

```
#:> myindex
#:> 0    1.1
#:> 1    1.1
#:> 2    1.1
#:> dtype: float64
```

Observe, sum on **string will concatenate column-wise**, whereas row-wise only sum up numeric fields

```
df.sum(0)
```

```
#:> One  X    3.30
#:>      Y    3.60
#:> Two  X    3.33
#:>      Y    3.66
#:> dtype: float64
```

```
df.sum(1)
```

```
#:> myindex
#:> 0    4.63
#:> 1    4.63
#:> 2    4.63
#:> dtype: float64
```

13.8.8 Plotting

13.9 class: Categories

13.9.1 Creating

13.9.1.1 From List

Basic (Auto Category Mapping)
Basic syntax return categorical index with sequence with code 0,1,2,3... mapping to first found category
In this case, **low(0), high(1), medium(2)**

```
temp = ['low','high','medium','high','high','low','medium','medium','high']
temp_cat = pd.Categorical(temp)
temp_cat
```

```
#:> ['low', 'high', 'medium', 'high', 'high', 'low', 'medium', 'medium', 'high']
#:> Categories (3, object): ['high', 'low', 'medium']
```

```
type( temp_cat )
```

```
#:> <class 'pandas.core.arrays.categorical.Categorical'>
```

Manual Category Mapping
During creation, we can specify mapping of codes to category: **low(0), medium(1), high(2)**

```
temp_cat = pd.Categorical(temp, categories=['low','medium','high'])
temp_cat
```

```
#:> ['low', 'high', 'medium', 'high', 'high', 'low', 'medium', 'medium', 'high']
#:> Categories (3, object): ['low', 'medium', 'high']
```

13.9.1.2 From Series

- We can 'add' categorical structure into a Series. With these methods, additional property (.cat) is added as a **categorical accessor**

- Through this accessor, you gain access to various properties of the category such as .codes, .categories. But not .get_values() as the information is in the Series itself

- Can we manual map category ?????

```
temp = ['low','high','medium','high','high','low','medium','medium','high']
temp_cat = pd.Series(temp, dtype='category')
print (type(temp_cat))       # Series object
```

```
#:> <class 'pandas.core.series.Series'>
```

```
print (type(temp_cat.cat))   # Categorical Accessor
```

```
#:> <class 'pandas.core.arrays.categorical.CategoricalAccessor'>
```

- Method below has the same result as above by using **.astype('category')**

- It is useful adding category structure into existing series.

```
temp_ser = pd.Series(temp)
temp_cat = pd.Series(temp).astype('category')
print (type(temp_cat))       # Series object
```

```
#:> <class 'pandas.core.series.Series'>
```

```
print (type(temp_cat.cat))   # Categorical Accessor
```

```
#:> <class 'pandas.core.arrays.categorical.CategoricalAccessor'>
```

```
temp_cat.cat.categories
```

```
#:> Index(['high', 'low', 'medium'], dtype='object')
```

13.9.1.3 Ordering Category

```
temp = ['low','high','medium','high','high','low','medium','medium','high']
temp_cat = pd.Categorical(temp, categories=['low','medium','high'], ordered=True)
temp_cat
```

```
#:> ['low', 'high', 'medium', 'high', 'high', 'low', 'medium', 'medium', 'high']
#:> Categories (3, object): ['low' < 'medium' < 'high']
```

```
# error
temp_cat.get_values()
```

```
temp_cat.codes
```

```
#:> array([0, 2, 1, 2, 2, 0, 1, 1, 2], dtype=int8)
```

```
temp_cat[0] < temp_cat[3]
```

```
#:> False
```

13.9.2 Properties

13.9.2.1 .categories

first element's code = 0
second element's code = 1
third element's code = 2

```
temp_cat.categories
```

```
#:> Index(['low', 'medium', 'high'], dtype='object')
```

13.9.2.2 .codes

Codes are actual **integer** value stored as array. 1 represent 'high',

```
temp_cat.codes
```

```
#:> array([0, 2, 1, 2, 2, 0, 1, 1, 2], dtype=int8)
```

13.9.3 Rename Category

13.9.3.1 Renamce To New Category Object

.rename_categories() method return a new category object with new changed categories

```
temp = ['low','high','medium','high','high','low','medium','medium','high']
new_temp_cat = temp_cat.rename_categories(['sejuk','sederhana','panas'])
new_temp_cat
```

```
#:> ['sejuk', 'panas', 'sederhana', 'panas', 'panas', 'sejuk', 'sederhana', 'sederhana', 'panas']
#:> Categories (3, object): ['sejuk' < 'sederhana' < 'panas']
```

```
temp_cat   # original category object categories not changed
```

```
#:> ['low', 'high', 'medium', 'high', 'high', 'low', 'medium', 'medium', 'high']
#:> Categories (3, object): ['low' < 'medium' < 'high']
```

13.9.3.2 Rename Inplace

Observe the original categories had been changed using **.rename()**

```
temp_cat.categories = ['sejuk','sederhana','panas']
temp_cat   # original category object categories is changed
```

```
#:> ['sejuk', 'panas', 'sederhana', 'panas', 'panas', 'sejuk', 'sederhana', 'sederhana
#:> Categories (3, object): ['sejuk' < 'sederhana' < 'panas']
```

13.9.4 Adding New Category

This return a new category object with added categories

```
temp_cat_more = temp_cat.add_categories(['susah','senang'])
temp_cat_more
```

```
#:> ['sejuk', 'panas', 'sederhana', 'panas', 'panas', 'sejuk', 'sederhana', 'sederhana
#:> Categories (5, object): ['sejuk' < 'sederhana' < 'panas' < 'susah' < 'senang']
```

13.9.5 Removing Category

This is **not in place**, hence return a new categorical object

13.9.5.1 Remove Specific Categor(ies)

Elements with its category removed will become **NaN**

```
temp = ['low','high','medium','high','high','low','medium','medium','high']
temp_cat = pd.Categorical(temp)
temp_cat_removed = temp_cat.remove_categories('low')
temp_cat_removed
```

```
#:> [NaN, 'high', 'medium', 'high', 'high', NaN, 'medium', 'medium', 'high']
#:> Categories (2, object): ['high', 'medium']
```

13.9.5.2 Remove Unused Category

Since categories removed are not used, there is no impact to the element

```
print (temp_cat_more)
```

```
#:> ['sejuk', 'panas', 'sederhana', 'panas', 'panas', 'sejuk', 'sederhana', 'sederhana
#:> Categories (5, object): ['sejuk' < 'sederhana' < 'panas' < 'susah' < 'senang']
```

```
temp_cat_more.remove_unused_categories()
```

```
#:> ['sejuk', 'panas', 'sederhana', 'panas', 'panas', 'sejuk', 'sederhana', 'sederhana
#:> Categories (3, object): ['sejuk' < 'sederhana' < 'panas']
```

13.9.6 Add and Remove Categories In One Step - Set()

```
temp = ['low','high','medium','high','high','low','medium','medium','high']
temp_cat = pd.Categorical(temp, ordered=True)
temp_cat
```

```
#:> ['low', 'high', 'medium', 'high', 'high', 'low', 'medium', 'medium', 'high']
#:> Categories (3, object): ['high' < 'low' < 'medium']
```

```
temp_cat.set_categories(['low','medium','sederhana','susah','senang'])
```

```
#:> ['low', NaN, 'medium', NaN, NaN, 'low', 'medium', 'medium', NaN]
#:> Categories (5, object): ['low' < 'medium' < 'sederhana' < 'susah' < 'senang']
```

13.9.7 Categorical Descriptive Analysis

13.9.7.1 At One Glance

```
temp_cat.describe()
```

```
#:>             counts     freqs
#:> categories
#:> high             4  0.444444
#:> low              2  0.222222
#:> medium           3  0.333333
```

13.9.7.2 Frequency Count

```
temp_cat.value_counts()
```

```
#:> high      4
#:> low       2
#:> medium    3
#:> dtype: int64
```

13.9.7.3 Least Frequent Category, Most Frequent Category, and Most Frequent Category

```
( temp_cat.min(), temp_cat.max(), temp_cat.mode() )
```

```
#:> ('high', 'medium', ['high']
#:> Categories (3, object): ['high' < 'low' < 'medium'])
```

13.9.8 Other Methods

13.9.8.1 .get_values()

Since actual value stored by categorical object are integer **codes**, get_values() function return values translated from *.codes** property

```
temp_cat.get_values()  #array
```

13.10 Dummies

- **get_dummies** creates columns for each categories
- The underlying data can be string or pd.Categorical

- It produces a **new pd.DataFrame**

13.10.1 Sample Data

```
df = pd.DataFrame (
    {'A': ['A1', 'A2', 'A3','A1','A3','A1'],
     'B': ['B1','B2','B3','B1','B1','B3'],
     'C': ['C1','C2','C3','C1',np.nan,np.nan]})
df
```

```
#:>     A   B    C
#:> 0  A1  B1   C1
#:> 1  A2  B2   C2
#:> 2  A3  B3   C3
#:> 3  A1  B1   C1
#:> 4  A3  B1  NaN
#:> 5  A1  B3  NaN
```

13.10.2 Dummies on Array-Like Data

```
pd.get_dummies(df.A)
```

```
#:>    A1  A2  A3
#:> 0   1   0   0
#:> 1   0   1   0
#:> 2   0   0   1
#:> 3   1   0   0
#:> 4   0   0   1
#:> 5   1   0   0
```

13.10.3 Dummies on DataFrame (multiple columns)

13.10.3.1 All Columns

```
pd.get_dummies(df)
```

```
#:>    A_A1  A_A2  A_A3  B_B1  B_B2  B_B3  C_C1  C_C2  C_C3
#:> 0     1     0     0     1     0     0     1     0     0
#:> 1     0     1     0     0     1     0     0     1     0
#:> 2     0     0     1     0     0     1     0     0     1
#:> 3     1     0     0     1     0     0     1     0     0
#:> 4     0     0     1     1     0     0     0     0     0
#:> 5     1     0     0     0     0     1     0     0     0
```

13.10.3.2 Selected Columns

```
cols = ['A','B']
pd.get_dummies(df[cols])
```

```
#:>    A_A1  A_A2  A_A3  B_B1  B_B2  B_B3
#:> 0     1     0     0     1     0     0
#:> 1     0     1     0     0     1     0
#:> 2     0     0     1     0     0     1
#:> 3     1     0     0     1     0     0
#:> 4     0     0     1     1     0     0
#:> 5     1     0     0     0     0     1
```

13.10.4 Dummies with na

By default, nan values are ignored

```
pd.get_dummies(df.C)
```

```
#:>    C1  C2  C3
#:> 0   1   0   0
#:> 1   0   1   0
#:> 2   0   0   1
#:> 3   1   0   0
#:> 4   0   0   0
#:> 5   0   0   0
```

Make NaN as a dummy variable

```
pd.get_dummies(df.C,dummy_na=True)
```

```
#:>    C1  C2  C3  NaN
#:> 0   1   0   0    0
#:> 1   0   1   0    0
```

```
#:> 2   0   0   1    0
#:> 3   1   0   0    0
#:> 4   0   0   0    1
#:> 5   0   0   0    1
```

13.10.5 Specify Prefixes

```
pd.get_dummies(df.A, prefix='col')
```

```
#:>    col_A1  col_A2  col_A3
#:> 0       1       0       0
#:> 1       0       1       0
#:> 2       0       0       1
#:> 3       1       0       0
#:> 4       0       0       1
#:> 5       1       0       0
```

```
pd.get_dummies(df[cols], prefix=['colA','colB'])
```

```
#:>    colA_A1  colA_A2  colA_A3  colB_B1  colB_B2  colB_B3
#:> 0        1        0        0        1        0        0
#:> 1        0        1        0        0        1        0
#:> 2        0        0        1        0        0        1
#:> 3        1        0        0        1        0        0
#:> 4        0        0        1        1        0        0
#:> 5        1        0        0        0        0        1
```

13.10.6 Dropping First Column

- Dummies cause **colinearity issue** for regression as it has redundant column.
- Dropping a column **does not loose any information** technically

```
pd.get_dummies(df[cols],drop_first=True)
```

```
#:>    A_A2  A_A3  B_B2  B_B3
#:> 0     0     0     0     0
#:> 1     1     0     1     0
#:> 2     0     1     0     1
#:> 3     0     0     0     0
#:> 4     0     1     0     0
#:> 5     0     0     0     1
```

13.11 DataFrameGroupBy

- `groupby()` is a DataFrame method, it returns **`DataFrameGroupBy`** object
- **`DataFrameGroupBy`** object open doors for dataframe aggregation and summarization
- **`DataFrameGroupBy`** object is a **very flexible abstraction**. In many ways, you can simply treat `DataFrameGroup` as if it's a **collection of DataFrames**, and it does the difficult things under the hood

13.11.1 Sample Data

```
company = pd.read_csv('data/company.csv')
company
```

```
#:>    Company Department      Name  Age  Salary  Birthdate
#:> 0       C1         D1      Yong   45   15000   1/1/1970
#:> 1       C1         D1      Chew   35   12000   2/1/1980
#:> 2       C1         D2       Lim   34    8000  2/19/1977
#:> 3       C1         D3     Jessy   23    2500  3/15/1990
#:> 4       C1         D3  Hoi Ming   55   25000  4/15/1987
#:> ..     ...        ...       ...  ...     ...        ...
#:> 13      C3         D3     Chang   32    7900  7/26/1973
#:> 14      C3         D1       Ong   44   17500  8/21/1980
#:> 15      C3         D2      Lily   41   15300  7/17/1990
#:> 16      C3         D3     Sally   54   21000  7/19/1968
#:> 17      C3         D3    Esther   37   13500  3/16/1969
#:>
#:> [18 rows x 6 columns]
```

13.11.2 Creating Groups

Group can be created for **single or multiple** columns

```
com_grp = company.groupby('Company') ## Single Column
com_dep_grp = company.groupby(['Company','Department'])  ## Multiple Column
type(com_dep_grp)
```

```
#:> <class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

13.11.3 Properties

13.11.3.1 Number of Groups

```
com_dep_grp.ngroups
```

```
#:> 9
```

13.11.3.2 Row Numbers Associated For Each Group

`.groups` property is a dictionary containing group key (identifying the group) and its values (underlying row indexes for the group)

```
gdict = com_dep_grp.groups        # return Dictionary
print( gdict.keys()   , '\n\n',   # group identifier
       gdict.values()   )         # group row indexes
```

```
#:> dict_keys([('C1', 'D1'), ('C1', 'D2'), ('C1', 'D3'), ('C2', 'D1'), ('C2', 'D2'), (
#:>
#:>  dict_values([Int64Index([0, 1], dtype='int64'), Int64Index([2], dtype='int64'), In
```

13.11.4 Methods

13.11.4.1 Number of Rows In Each Group

```
com_dep_grp.size()  # return panda Series object
```

```
#:> Company  Department
#:> C1       D1            2
#:>          D2            1
#:>          D3            3
#:> C2       D1            1
#:>          D2            3
#:>          D3            3
#:> C3       D1            1
#:>          D2            1
#:>          D3            3
#:> dtype: int64
```

13.11.5 Retrieve Rows

13.11.5.1 Retrieve n-th Row Of Each Grou

- Row number is 0-based
- For First row, use `.first()` or `nth(0)`

```
print( com_dep_grp.nth(0)  , '\n',
       com_dep_grp.first())
```

```
#:>                            Name  Age  Salary   Birthdate
#:> Company Department
#:> C1      D1                 Yong   45   15000    1/1/1970
#:>         D2                  Lim   34    8000   2/19/1977
```

```
#:>         D3           Jessy   23    2500   3/15/1990
#:> C2      D1            Anne   18     400   7/15/1997
#:>         D2         Deborah   30    8600   8/15/1984
#:>         D3         Michael   38   17000  11/30/1997
#:> C3      D1             Ong   44   17500   8/21/1980
#:>         D2            Lily   41   15300   7/17/1990
#:>         D3           Chang   32    7900   7/26/1973
#:>                       Name  Age  Salary   Birthdate
#:> Company Department
#:> C1      D1            Yong   45   15000    1/1/1970
#:>         D2             Lim   34    8000   2/19/1977
#:>         D3           Jessy   23    2500   3/15/1990
#:> C2      D1            Anne   18     400   7/15/1997
#:>         D2         Deborah   30    8600   8/15/1984
#:>         D3         Michael   38   17000  11/30/1997
#:> C3      D1             Ong   44   17500   8/21/1980
#:>         D2            Lily   41   15300   7/17/1990
#:>         D3           Chang   32    7900   7/26/1973
```

- For Last row, use `.last()` or `nth`(-1)‘

```
print( com_dep_grp.nth(-1)  , '\n',
       com_dep_grp.last())
```

```
#:>                      Name  Age  Salary   Birthdate
#:> Company Department
#:> C1      D1           Chew   35   12000    2/1/1980
#:>         D2            Lim   34    8000   2/19/1977
#:>         D3        Sui Wei   56    3000   6/15/1990
#:> C2      D1           Anne   18     400   7/15/1997
#:>         D2          Jimmy   46   14000  10/31/1988
#:>         D3        Bernard   29    9800   12/1/1963
#:> C3      D1            Ong   44   17500   8/21/1980
#:>         D2           Lily   41   15300   7/17/1990
#:>         D3         Esther   37   13500   3/16/1969
#:>                       Name  Age  Salary   Birthdate
#:> Company Department
#:> C1      D1            Chew   35   12000    2/1/1980
#:>         D2             Lim   34    8000   2/19/1977
#:>         D3         Sui Wei   56    3000   6/15/1990
#:> C2      D1            Anne   18     400   7/15/1997
#:>         D2           Jimmy   46   14000  10/31/1988
#:>         D3         Bernard   29    9800   12/1/1963
#:> C3      D1             Ong   44   17500   8/21/1980
#:>         D2            Lily   41   15300   7/17/1990
#:>         D3          Esther   37   13500   3/16/1969
```

13.11.5.2 Retrieve N Rows Of Each Groups

Example below retrieve 2 rows from each group

```
com_dep_grp.head(2)
```

```
#:>    Company Department      Name  Age  Salary   Birthdate
#:> 0       C1         D1      Yong   45   15000    1/1/1970
#:> 1       C1         D1      Chew   35   12000    2/1/1980
#:> 2       C1         D2       Lim   34    8000   2/19/1977
#:> 3       C1         D3     Jessy   23    2500   3/15/1990
#:> 4       C1         D3  Hoi Ming   55   25000   4/15/1987
#:> ..     ...        ...       ...  ...     ...         ...
#:> 11      C2         D3   Jeannie   30   12500  12/31/1980
#:> 13      C3         D3     Chang   32    7900   7/26/1973
#:> 14      C3         D1       Ong   44   17500   8/21/1980
#:> 15      C3         D2      Lily   41   15300   7/17/1990
#:> 16      C3         D3     Sally   54   21000   7/19/1968
#:>
#:> [14 rows x 6 columns]
```

13.11.5.3 Retrieve All Rows Of Specific Group

`get_group()` retrieves all rows within the specified group.

```
com_dep_grp.get_group(('C1','D3'))
```

```
#:>   Company Department      Name  Age  Salary  Birthdate
#:> 3      C1         D3     Jessy   23    2500  3/15/1990
#:> 4      C1         D3  Hoi Ming   55   25000  4/15/1987
#:> 5      C1         D3   Sui Wei   56    3000  6/15/1990
```

13.11.6 Single Statistic Per Group

13.11.6.1 count()

`count()` for valid data (not null) for each fields within the group

```
com_dep_grp.count()  # return panda DataFrame object
```

```
#:>                     Name  Age  Salary  Birthdate
#:> Company Department
#:> C1      D1             2    2       2          2
#:>         D2             1    1       1          1
#:>         D3             3    3       3          3
#:> C2      D1             1    1       1          1
#:>         D2             3    3       3          3
#:>         D3             3    3       3          3
#:> C3      D1             1    1       1          1
```

```
#:>         D2              1    1       1           1
#:>         D3              3    3       3           3
```

13.11.6.2 sum()

This sums up all numeric columns for each group

```
com_dep_grp.sum()
```

```
#:>                     Age  Salary
#:> Company Department
#:> C1      D1           80   27000
#:>         D2           34    8000
#:>         D3          134   30500
#:> C2      D1           18     400
#:>         D2          127   34600
#:>         D3           97   39300
#:> C3      D1           44   17500
#:>         D2           41   15300
#:>         D3          123   42400
```

To sum specific columns of each group, use `['columnName']` to select the column. When single column is selected, output is a **Series**

```
com_dep_grp['Age'].sum()
```

```
#:> Company  Department
#:> C1       D1             80
#:>          D2             34
#:>          D3            134
#:> C2       D1             18
#:>          D2            127
#:>          D3             97
#:> C3       D1             44
#:>          D2             41
#:>          D3            123
#:> Name: Age, dtype: int64
```

13.11.6.3 mean()

This average up all numeric columns for each group

```
com_dep_grp.mean()
```

```
#:>                           Age        Salary
#:> Company Department
#:> C1      D1          40.000000  13500.000000
#:>         D2          34.000000   8000.000000
#:>         D3          44.666667  10166.666667
```

```
#:> C2      D1          18.000000    400.000000
#:>         D2          42.333333  11533.333333
#:>         D3          32.333333  13100.000000
#:> C3      D1          44.000000  17500.000000
#:>         D2          41.000000  15300.000000
#:>         D3          41.000000  14133.333333
```

To average specific columns of each group, use `['columnName']` to select the column.
When single column is selected, output is a **Series**

```
com_dep_grp['Age'].mean()
```

```
#:> Company  Department
#:> C1       D1            40.000000
#:>          D2            34.000000
#:>          D3            44.666667
#:> C2       D1            18.000000
#:>          D2            42.333333
#:>          D3            32.333333
#:> C3       D1            44.000000
#:>          D2            41.000000
#:>          D3            41.000000
#:> Name: Age, dtype: float64
```

13.11.7 Multi Statistic Per Group

13.11.7.1 Single Function To Column(s)

- Instructions for aggregation are provided in the form of a dictionary. Dictionary keys specifies the **column name**, and value as the **function** to run
- Can use `lambda x:` to customize the calclulation on entire column (x)
- Python built-in function names does can be supplied without wrapping in string `'function'`

```
com_dep_grp.agg({
  'Age': sum ,                  ## Total age of the group
  'Salary': lambda x: max(x),  ## Highest salary of the group
  'Birthdate': 'first'          ## First birthday of the group
})
```

```
#:>                     Age  Salary   Birthdate
#:> Company Department
#:> C1      D1           80   15000    1/1/1970
#:>         D2           34    8000   2/19/1977
#:>         D3          134   25000   3/15/1990
```

```
#:> C2      D1           18     400   7/15/1997
#:>         D2          127   14000   8/15/1984
#:>         D3           97   17000  11/30/1997
#:> C3      D1           44   17500   8/21/1980
#:>         D2           41   15300   7/17/1990
#:>         D3          123   21000   7/26/1973
```

13.11.7.2 Multiple Function to Column(s)

- Use list of function names to specify functions to be applied on a particular column

- Notice that output columns are MultiIndex , indicating the name of funcitons appled on level 1

```
ag = com_dep_grp.agg({
      'Age': ['mean', sum ],       ## Average age of the group
      'Salary': lambda x: max(x),  ## Highest salary of the group
      'Birthdate': 'first'         ## First birthday of the group
    })

print (ag, '\n\n', ag.columns)
```

```
#:>                          Age        Salary   Birthdate
#:>                         mean  sum <lambda>       first
#:> Company Department
#:> C1      D1         40.000000   80    15000    1/1/1970
#:>         D2         34.000000   34     8000   2/19/1977
#:>         D3         44.666667  134    25000   3/15/1990
#:> C2      D1         18.000000   18      400   7/15/1997
#:>         D2         42.333333  127    14000   8/15/1984
#:>         D3         32.333333   97    17000  11/30/1997
#:> C3      D1         44.000000   44    17500   8/21/1980
#:>         D2         41.000000   41    15300   7/17/1990
#:>         D3         41.000000  123    21000   7/26/1973
#:>
#:>  MultiIndex([(      'Age',     'mean'),
#:>             (      'Age',      'sum'),
#:>             (   'Salary', '<lambda>'),
#:>             ('Birthdate',    'first')],
#:>            )
```

13.11.7.3 Column Relabling

Introduced in Pandas 0.25.0, groupby aggregation with relabelling is supported using “named aggregation” with **simple tuples**

```
com_dep_grp.agg(
  max_age     = ('Age', max),
  salary_m100 = ('Salary',  lambda x: max(x)+100),
  first_bd    = ('Birthdate', 'first')
)
```

```
#:>                     max_age  salary_m100    first_bd
#:> Company Department
#:> C1      D1               45        15100    1/1/1970
#:>         D2               34         8100   2/19/1977
#:>         D3               56        25100   3/15/1990
#:> C2      D1               18          500   7/15/1997
#:>         D2               51        14100   8/15/1984
#:>         D3               38        17100  11/30/1997
#:> C3      D1               44        17600   8/21/1980
#:>         D2               41        15400   7/17/1990
#:>         D3               54        21100   7/26/1973
```

13.11.8 Iteration

DataFrameGroupBy object can be thought as a collection of named groups

```
def print_groups (g):
    for name,group in g:
        print (name)
        print (group[:2])

print_groups (com_grp)
```

```
#:> C1
#:>   Company Department  Name  Age  Salary Birthdate
#:> 0      C1         D1  Yong   45   15000  1/1/1970
#:> 1      C1         D1  Chew   35   12000  2/1/1980
#:> C2
#:>   Company Department     Name  Age  Salary  Birthdate
#:> 6      C2         D1     Anne   18     400  7/15/1997
#:> 7      C2         D2  Deborah   30    8600  8/15/1984
#:> C3
#:>    Company Department   Name  Age  Salary  Birthdate
#:> 13      C3         D3  Chang   32    7900  7/26/1973
#:> 14      C3         D1    Ong   44   17500  8/21/1980
```

```
com_grp
```

```
#:> <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f9f6decc3d0>
```

13.11.9 Transform

- Transform is an operation used combined with **DataFrameGroupBy** object
- **transform()** return a **new DataFrame object**

```
grp = company.groupby('Company')
grp.size()
```

```
#:> Company
#:> C1    6
#:> C2    7
#:> C3    5
#:> dtype: int64
```

transform() perform a function to a group, and **expands and replicate** it to multiple rows according to original DataFrame

```
grp[['Age','Salary']].transform('sum')
```

```
#:>     Age  Salary
#:> 0   248   65500
#:> 1   248   65500
#:> 2   248   65500
#:> 3   248   65500
#:> 4   248   65500
#:> ..  ...     ...
#:> 13  208   75200
#:> 14  208   75200
#:> 15  208   75200
#:> 16  208   75200
#:> 17  208   75200
#:>
#:> [18 rows x 2 columns]
```

```
grp.transform( lambda x:x+10 )
```

```
#:>     Age  Salary
#:> 0    55   15010
#:> 1    45   12010
#:> 2    44    8010
#:> 3    33    2510
#:> 4    65   25010
#:> ..  ...     ...
#:> 13   42    7910
#:> 14   54   17510
#:> 15   51   15310
#:> 16   64   21010
```

```
#:> 17   47   13510
#:>
#:> [18 rows x 2 columns]
```

13.12 Fundamental Analysis

13.13 Missing Data

13.13.1 What Is Considered Missing Data ?

13.13.2 Sample Data

```
df = pd.DataFrame( np.random.randn(5, 3),
                   index   =['a', 'c', 'e', 'f', 'h'],
                   columns =['one', 'two', 'three'])
df['four'] = 'bar'
df['five'] = df['one'] > 0
#df
df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
```

```
#:>         one       two     three four   five
#:> a -0.155909 -0.501790  0.235569  bar  False
#:> b       NaN       NaN       NaN  NaN    NaN
#:> c -1.763605 -1.095862 -1.087766  bar  False
#:> d       NaN       NaN       NaN  NaN    NaN
#:> e -0.305170 -0.473748 -0.200595  bar  False
#:> f  0.355197  0.689518  0.410590  bar   True
#:> g       NaN       NaN       NaN  NaN    NaN
#:> h -0.564978  0.599391 -0.162936  bar  False
```

How Missing Data For Each Column ?

```
df.count()
```

```
#:> one      5
#:> two      5
#:> three    5
#:> four     5
#:> five     5
#:> dtype: int64
```

```
len(df.index) - df.count()
```

```
#:> one      0
#:> two      0
#:> three    0
#:> four     0
```

```
#:> five     0
#:> dtype: int64
```

```
df.isnull()
```

```
#:>      one    two  three   four   five
#:> a  False  False  False  False  False
#:> c  False  False  False  False  False
#:> e  False  False  False  False  False
#:> f  False  False  False  False  False
#:> h  False  False  False  False  False
```

```
df.describe()
```

```
#:>             one       two     three
#:> count  5.000000  5.000000  5.000000
#:> mean  -0.486893 -0.156498 -0.161028
#:> std    0.788635  0.772882  0.579752
#:> min   -1.763605 -1.095862 -1.087766
#:> 25%   -0.564978 -0.501790 -0.200595
#:> 50%   -0.305170 -0.473748 -0.162936
#:> 75%   -0.155909  0.599391  0.235569
#:> max    0.355197  0.689518  0.410590
```

Chapter 14

matplotlib

14.1 Library

```
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

from plydata import define, query, select, group_by, summarize, arrange, head, rename
import plotnine
from plotnine import *
```

14.2 Sample Data

This chapter uses the sample data generate with below code. The idea is to simulate two categorical-alike feature, and two numeric value feature:

- com is random character between ?C1?, ?C2? and ?C3?
- dept is random character between ?D1?, ?D2?, ?D3?, ?D4? and ?D5?
- grp is random character with randomly generated ?G1?, ?G2?
- value1 represents numeric value, normally distributed at mean 50
- value2 is numeric value, normally distributed at mean 25

```
n = 200
comp = ['C' + i for i in np.random.randint( 1,4, size  = n).astype(str)] # 3x Company
dept = ['D' + i for i in np.random.randint( 1,6, size  = n).astype(str)] # 5x Department
```

```
grp =  ['G' + i for i in np.random.randint( 1,3, size  = n).astype(str)] # 2x Groups
value1 = np.random.normal( loc=50 , scale=5 , size = n)
value2 = np.random.normal( loc=20 , scale=3 , size = n)
value3 = np.random.normal( loc=5 , scale=30 , size = n)

mydf = pd.DataFrame({
    'comp':comp,
    'dept':dept,
    'grp': grp,
    'value1':value1,
    'value2':value2,
    'value3':value3 })
mydf.head()
```

```
#:>   comp dept grp     value1     value2     value3
#:> 0   C1   D3  G2  46.654951  16.361565   5.829870
#:> 1   C1   D2  G1  52.078667  23.256426  -8.347119
#:> 2   C1   D1  G2  50.290409  21.142525 -25.180967
#:> 3   C1   D5  G1  60.976496  14.238000 -40.435388
#:> 4   C2   D5  G1  44.134974  23.528127   7.772335
```

```
mydf.info()
```

```
#:> <class 'pandas.core.frame.DataFrame'>
#:> RangeIndex: 200 entries, 0 to 199
#:> Data columns (total 6 columns):
#:>  #   Column  Non-Null Count  Dtype
#:> ---  ------  --------------  -----
#:>  0   comp    200 non-null    object
#:>  1   dept    200 non-null    object
#:>  2   grp     200 non-null    object
#:>  3   value1  200 non-null    float64
#:>  4   value2  200 non-null    float64
#:>  5   value3  200 non-null    float64
#:> dtypes: float64(3), object(3)
#:> memory usage: 9.5+ KB
```

14.3 MATLAB-like API

- The good thing about the pylab MATLAB-style API is that it is easy to get started with if you are familiar with MATLAB, and it has a minumum of coding overhead for simple plots.

- However, I'd encourrage not using the MATLAB compatible API for anything but the simplest figures.

- Instead, I recommend learning and using matplotlib's object-oriented plotting API. It is remarkably powerful. For advanced figures with subplots, insets and other components it is very nice to work with.

14.3.1 Sample Data

```
# Sample Data
x = np.linspace(0,5,10)
y = x ** 2
```

14.3.2 Single Plot

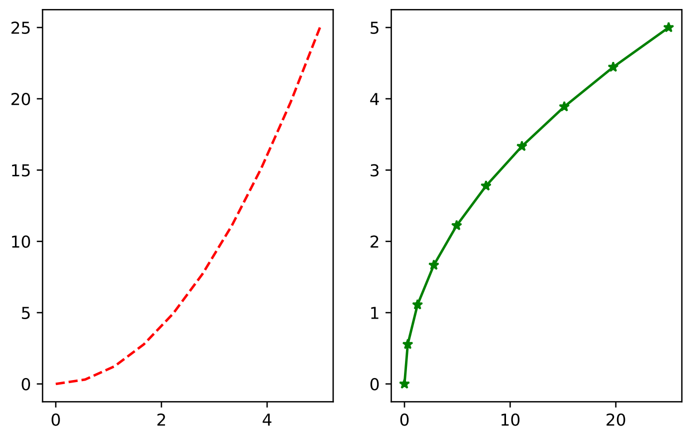
```
plt.figure()
plt.xlabel('x')
plt.ylabel('y')
plt.plot(x,y,'red')
plt.title('My Good Data')
plt.show()
```

14.3.3 Multiple Subplots

Each call lto **subplot()** will create a new container for subsequent plot command

```
plt.figure()
plt.subplot(1,2,1) # 1 row, 2 cols, at first box
plt.plot(x,y,'r--')
plt.subplot(1,2,2) # 1 row, 2 cols, at second box
```

```
plt.plot(y,x,'g*-')
plt.show()
```



14.4 Object-Oriented API

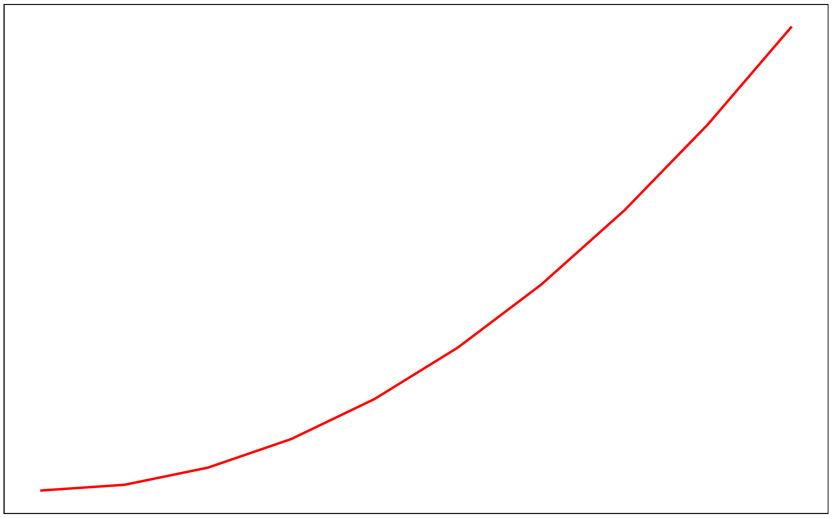
14.4.1 Sample Data

```
# Sample Data
x = np.linspace(0,5,10)
y = x ** 2
```

14.4.2 Single Plot

One figure, one axes

```
fig = plt.figure()
axes = fig.add_axes([0,0,1,1]) # left, bottom, width, height (range 0 to 1)
axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title')
plt.show()
```



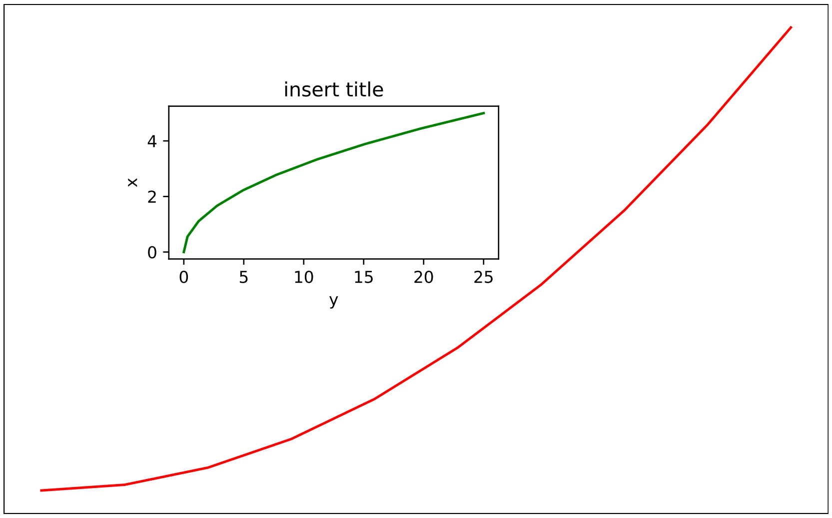
14.4.3 Multiple Axes In One Plot

- This is still considered a **single plot**, but with **multiple axes**

```
fig = plt.figure()
ax1 = fig.add_axes([0, 0, 1, 1])          # main axes
ax2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

ax1.plot(x,y,'r')
ax1.set_xlabel('x')
ax1.set_ylabel('y')

ax2.plot(y, x, 'g')
ax2.set_xlabel('y')
ax2.set_ylabel('x')
ax2.set_title('insert title')
plt.show()
```



14.4.4 Multiple Subplots

- One **figure** can contain multiple **subplots**
- Each subplot has **one axes**

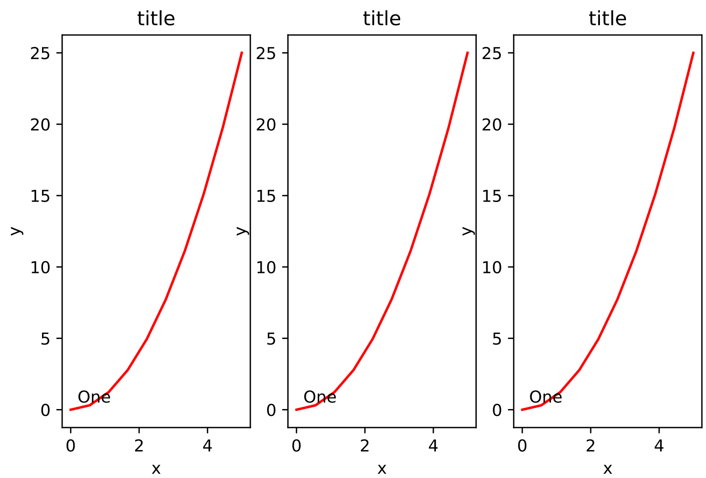
14.4.4.1 Simple Subplots - all same size

- subplots() function return axes object that is iterable.

Single Row Grid
Single row grid means axes is an 1-D array. Hence can use **for** to iterate through axes

```
fig, axes = plt.subplots( nrows=1,ncols=3 )
print (axes.shape)
```

```
for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')
    ax.text(0.2,0.5,'One')
plt.show()
```

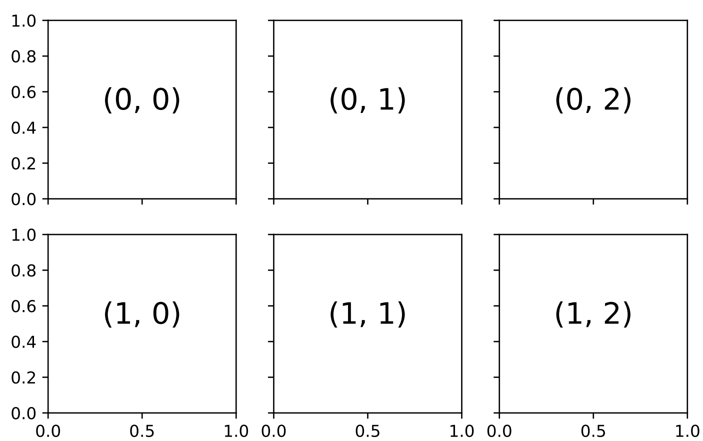


Multiple Row Grid

Multile row grid means axes is an 2-D array. Hence can use two levels of **for** loop to iterate through each row and column

```
fig, axes = plt.subplots(2, 3, sharex='col', sharey='row')
print (axes.shape)
```

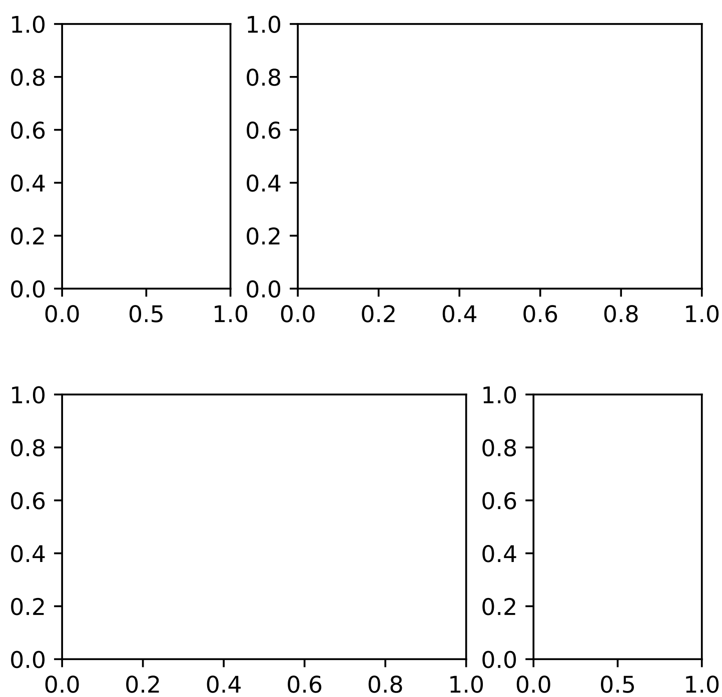
```
for i in range(axes.shape[0]):
    for j in range(axes.shape[1]):
        axes[i, j].text(0.5, 0.5, str((i, j)),
                        fontsize=18, ha='center')
plt.show()
```



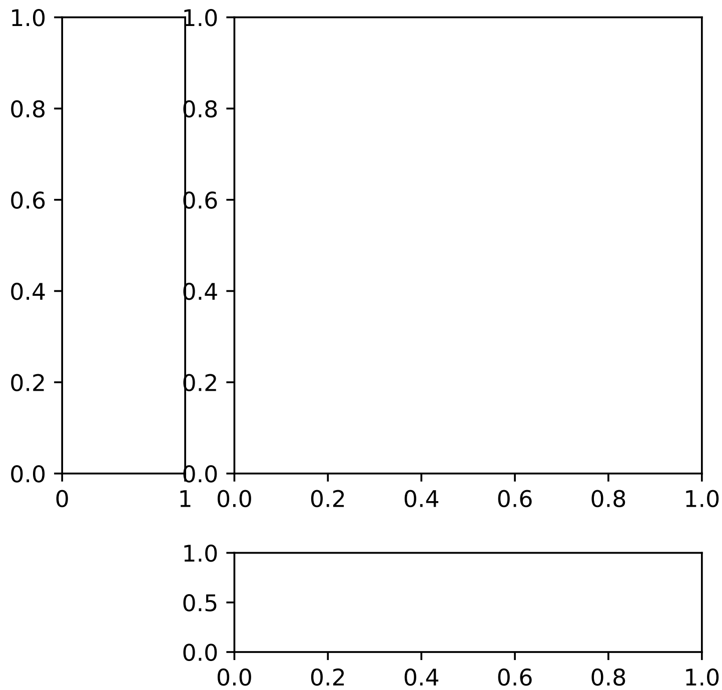
14.4.4.2 Complicated Subplots - different size

- **GridSpec** specify grid size of the figure
- Manually specify each subplot and their relevant grid position and size

```
plt.figure(figsize=(5,5))
grid = plt.GridSpec(2, 3, hspace=0.4, wspace=0.4)
plt.subplot(grid[0, 0])  #row 0, col 0
plt.subplot(grid[0, 1:]) #row 0, col 1 to :
plt.subplot(grid[1, :2]) #row 1, col 0:2
plt.subplot(grid[1, 2]); #row 1, col 2
plt.show()
```

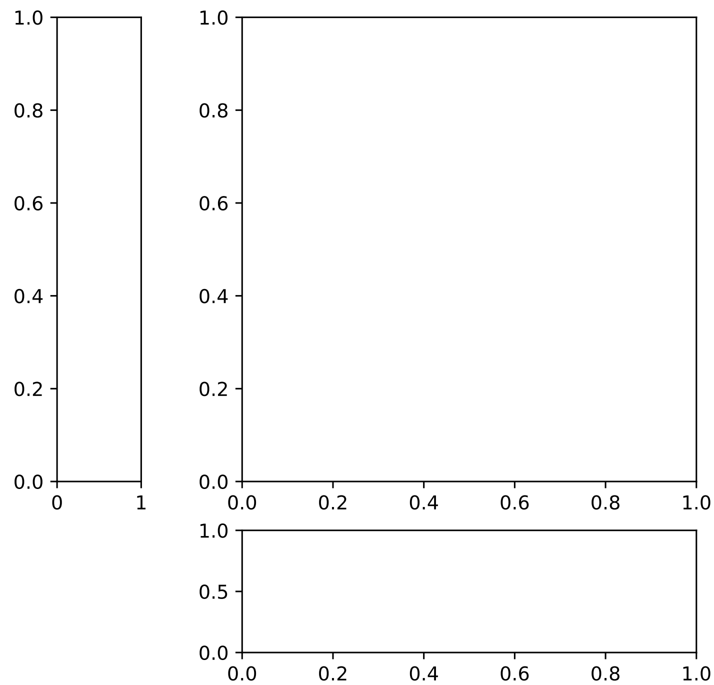


```
plt.figure(figsize=(5,5))
grid = plt.GridSpec(4, 4, hspace=0.8, wspace=0.4)
plt.subplot(grid[:3, 0])    # row 0:3, col 0
plt.subplot(grid[:3, 1: ])  # row 0:3, col 1:
plt.subplot(grid[3, 1: ]);  # row 3,   col 1:
plt.show()
```



-1 means last row or column

```
plt.figure(figsize=(6,6))
grid = plt.GridSpec(4, 4, hspace=0.4, wspace=1.2)
plt.subplot(grid[:-1, 0 ])  # row 0 till last row (not including last row), col 0
plt.subplot(grid[:-1, 1:])  # row 0 till last row (not including last row), col 1 till end
plt.subplot(grid[-1, 1: ]); # row last row, col 1 till end
plt.show()
```

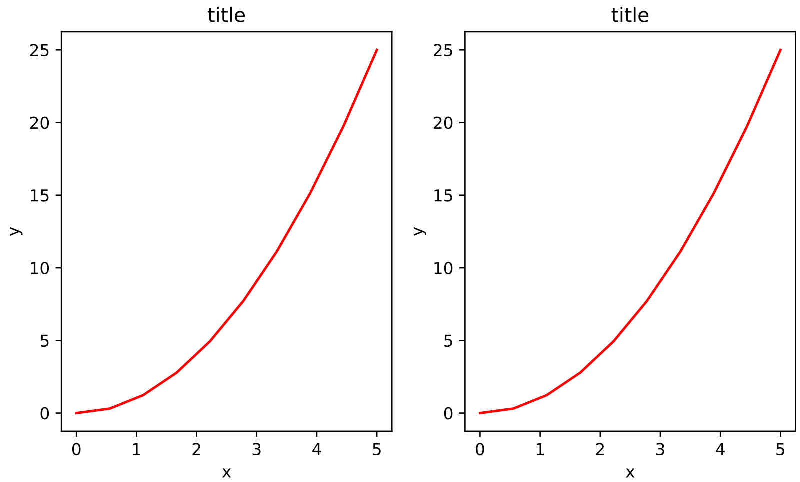


14.4.5 Figure Customization

14.4.5.1 Avoid Overlap - Use tight_layout()

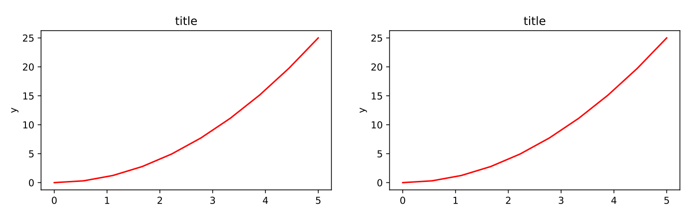
Sometimes when the figure size is too small, plots will overlap each other. - **tight_layout()** will introduce extra white space in between the subplots to avoid overlap.
- The figure became wider.

```
fig, axes = plt.subplots( nrows=1,ncols=2)
for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')
fig.tight_layout() # adjust the positions of axes so that there is no overlap
plt.show()
```



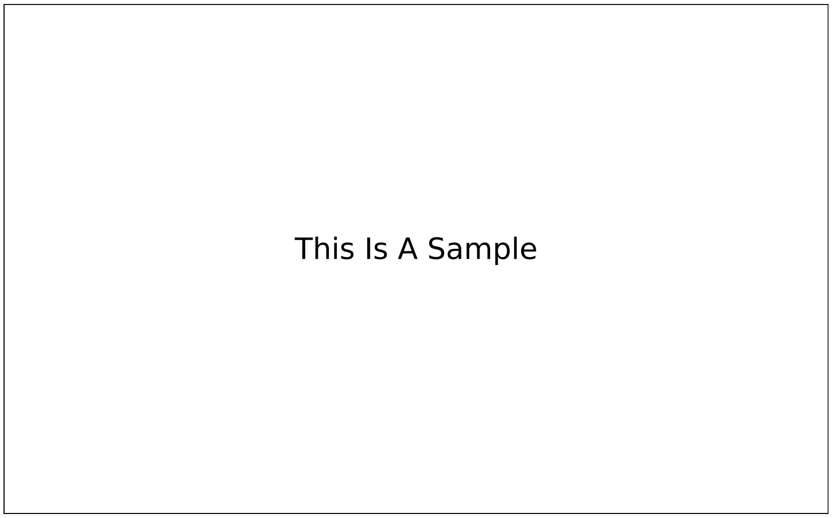
14.4.5.2 Avoid Overlap - Change Figure Size

```
fig, axes = plt.subplots( nrows=1,ncols=2,figsize=(12,3))
for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')
plt.show()
```



14.4.5.3 Text Within Figure

```
fig = plt.figure()
fig.text(0.5, 0.5, 'This Is A Sample',fontsize=18, ha='center');
axes = fig.add_axes([0,0,1,1]) # left, bottom, width, height (range 0 to 1)
plt.show()
```



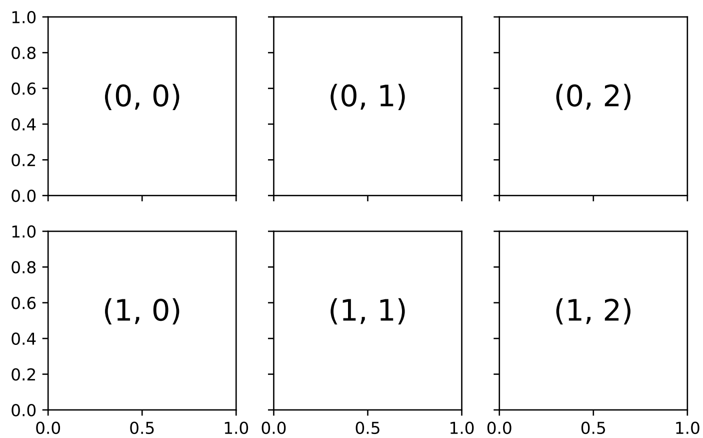
14.4.6 Axes Customization

14.4.6.1 Y-Axis Limit

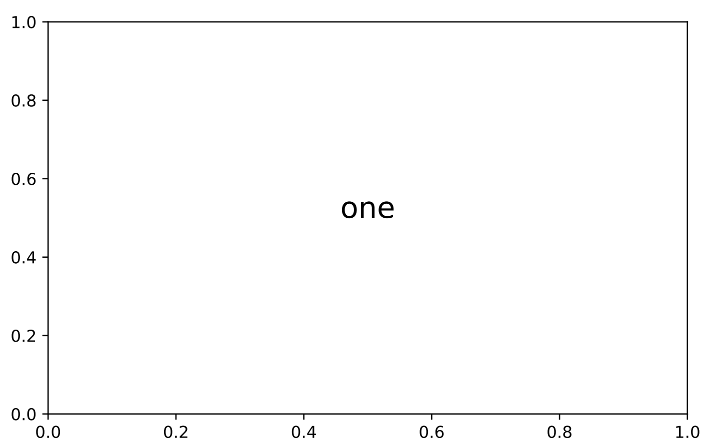
```
fig = plt.figure()
fig.add_axes([0,0,1,1], ylim=(-2,5));
plt.show()
```

14.4.6.2 Text Within Axes

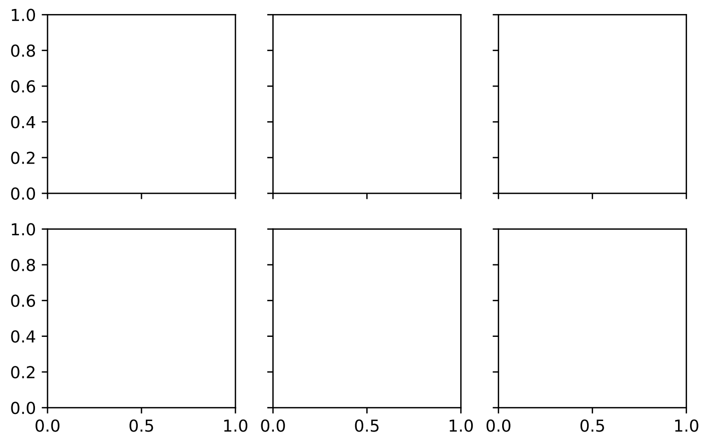
```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
for i in range(2):
    for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)),
                      fontsize=18, ha='center')
plt.show()
```




```
plt.text(0.5, 0.5, 'one',fontsize=18, ha='center')
plt.show()
```



```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row') # removed inner label
plt.show()
```

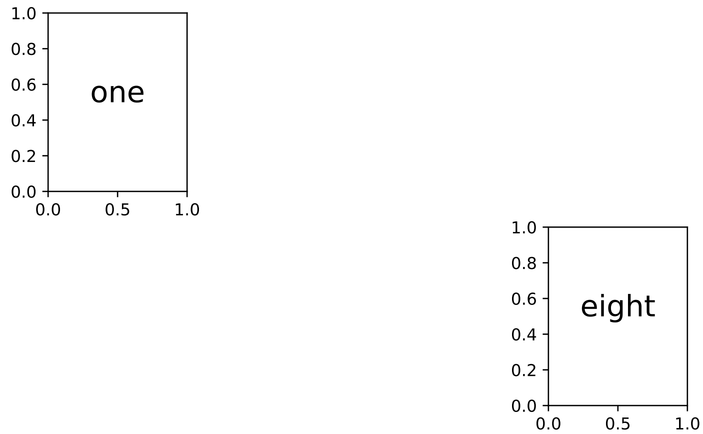


14.4.6.4 Create Subplot Individually

Each call lto **subplot()** will create a new container for subsequent plot command

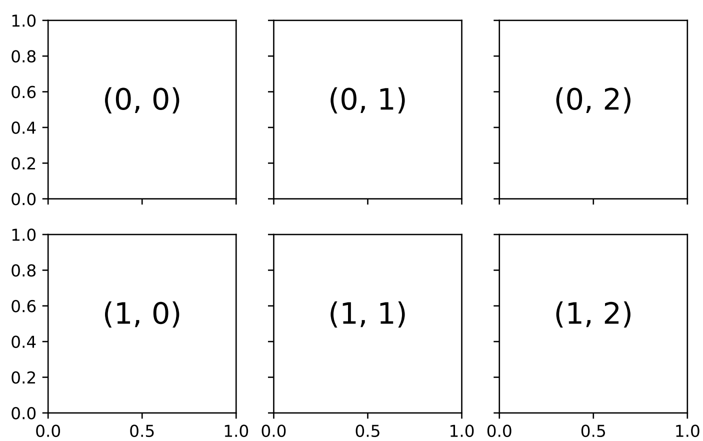
```
plt.subplot(2,4,1)
plt.text(0.5, 0.5, 'one',fontsize=18, ha='center')

plt.subplot(2,4,8)
plt.text(0.5, 0.5, 'eight',fontsize=18, ha='center')
plt.show()
```



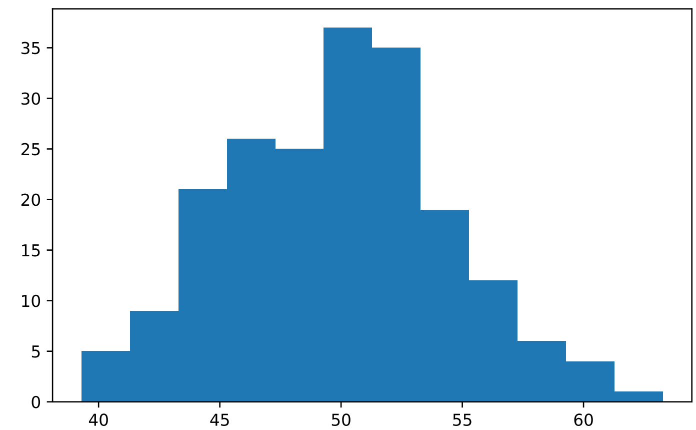
Iterate through subplots (ax) to populate them

```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
for i in range(2):
    for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)),
                      fontsize=18, ha='center')
plt.show()
```



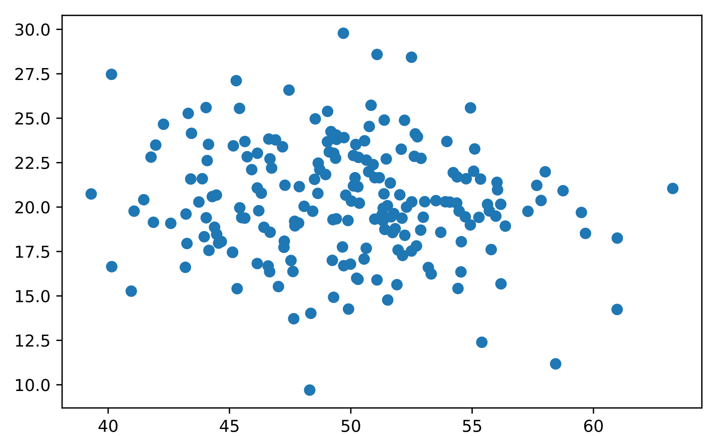
14.5 Histogram

```
plt.hist(mydf.value1, bins=12);
plt.show()
```



14.6 Scatter Plot

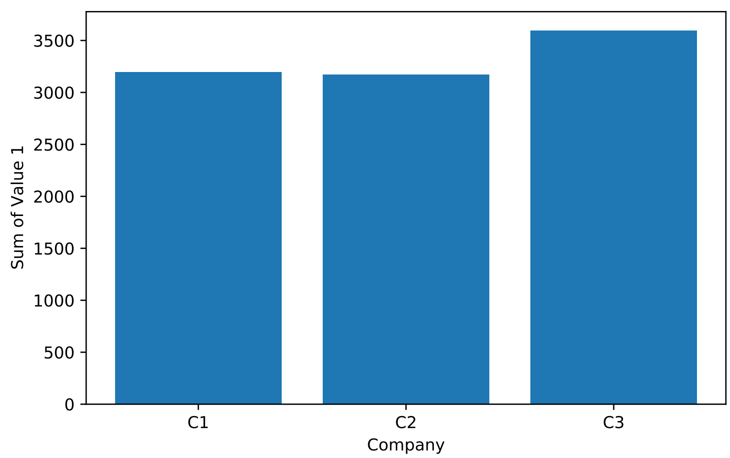
```
plt.scatter(mydf.value1, mydf.value2)
plt.show()
```



14.7 Bar Chart

```
com_grp = mydf.groupby('comp')
grpdf = com_grp['value1'].sum().reset_index()
grpdf
```

```
plt.bar(grpdf.comp, grpdf.value1);
plt.xlabel('Company')
plt.ylabel('Sum of Value 1')
plt.show()
```



Chapter 15

seaborn

15.1 Seaborn and Matplotlib

- seaborn **returns a matplotlib object** that can be modified by the options in the pyplot module
- Often, these options are wrapped by seaborn and .plot() in pandas and available as arguments

15.2 Sample Data

```
n = 100
comp = ['C' + i for i in np.random.randint( 1,4, size  = n).astype(str)] # 3x Company
dept = ['D' + i for i in np.random.randint( 1,4, size  = n).astype(str)] # 5x Department
grp =  ['G' + i for i in np.random.randint( 1,4, size  = n).astype(str)] # 2x Groups
value1 = np.random.normal( loc=50 , scale=5 , size = n)
value2 = np.random.normal( loc=20 , scale=3 , size = n)
value3 = np.random.normal( loc=5 , scale=30 , size = n)

mydf = pd.DataFrame({
    'comp':comp,
    'dept':dept,
    'grp': grp,
    'value1':value1,
    'value2':value2,
    'value3':value3
})
mydf.head()
```

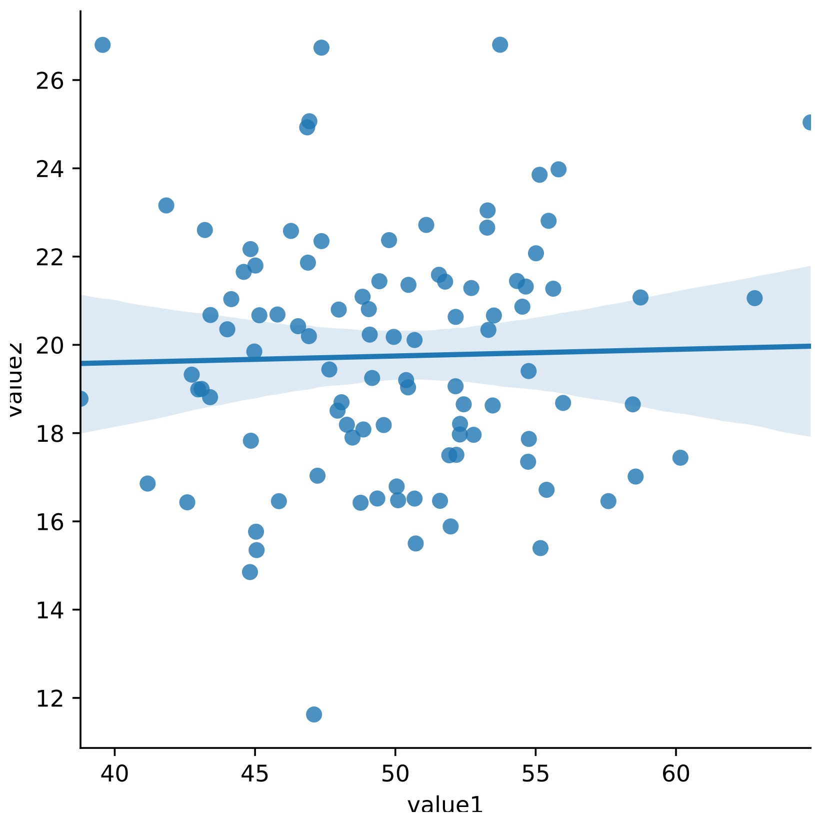
```
#:>   comp dept grp     value1     value2     value3
#:> 0   C2   D3  G1  52.435646  18.652742 -17.102075
#:> 1   C2   D2  G2  46.934847  25.066257  -7.635967
#:> 2   C2   D2  G2  51.101242  22.717363  -6.557646
#:> 3   C3   D1  G1  47.983460  20.799023  -0.952821
#:> 4   C2   D1  G1  49.173047  19.248937   1.231331
```

15.3 Scatter Plot

15.3.1 2x Numeric

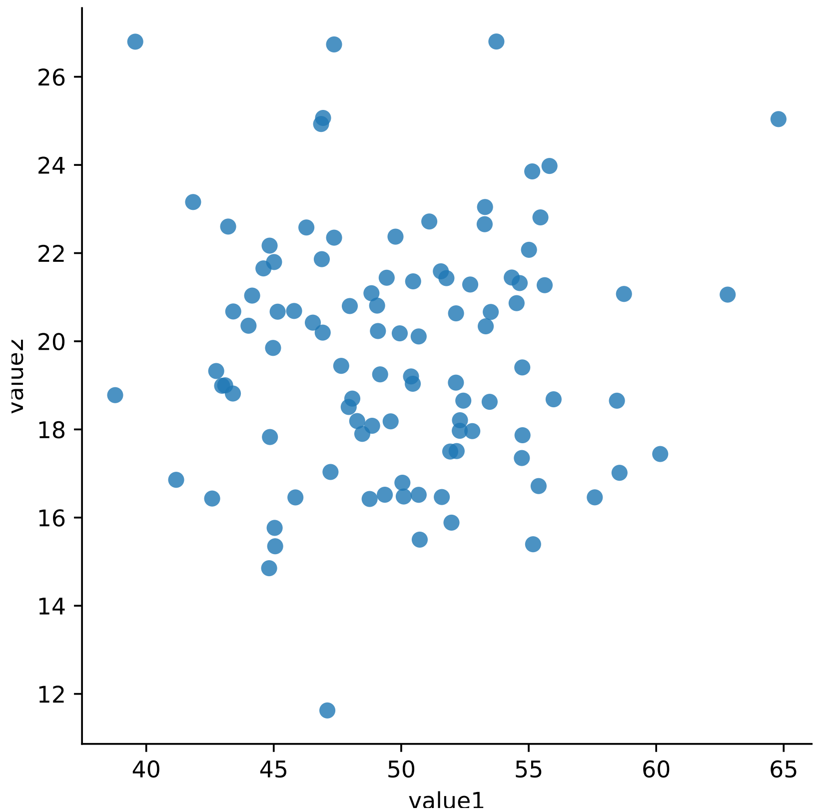
```
sns.lmplot(x='value1', y='value2', data=mydf)
```

```
plt.show()
```




```
sns.lmplot(x='value1', y='value2', fit_reg=False, data=mydf);  #hide regresion line
```

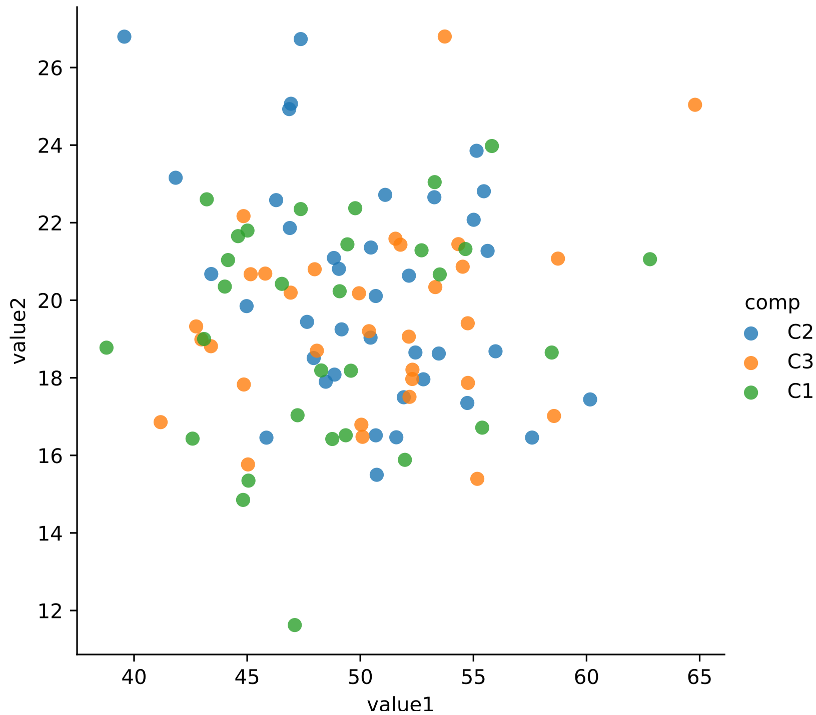
```
plt.show()
```



15.3.2 2xNumeric + 1x Categorical

Use **hue** to represent additional categorical feature

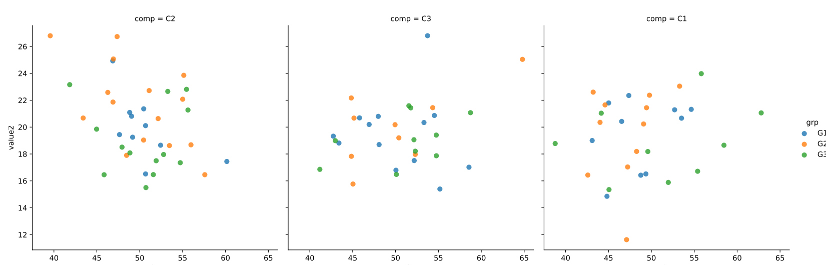
```
sns.lmplot(x='value1', y='value2', data=mydf, hue='comp', fit_reg=False);
plt.show()
```



15.3.3 2xNumeric + 2x Categorical

Use **col** and **hue** to represent two categorical features

```
sns.lmplot(x='value1', y='value2', col='comp',hue='grp', fit_reg=False, data=mydf);
plt.show()
```

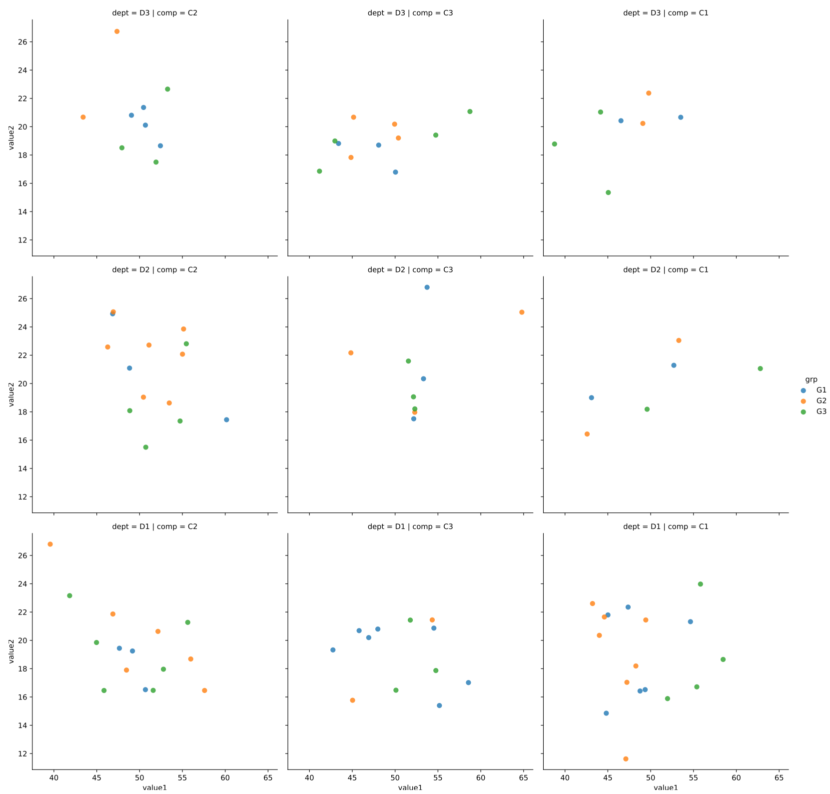


15.3.4 2xNumeric + 3x Categorical

Use **row, col** and **hue** to represent three categorical features

```
sns.lmplot(x='value1', y='value2', row='dept',col='comp', hue='grp', fit_reg=False, da
```

```
plt.show()
```



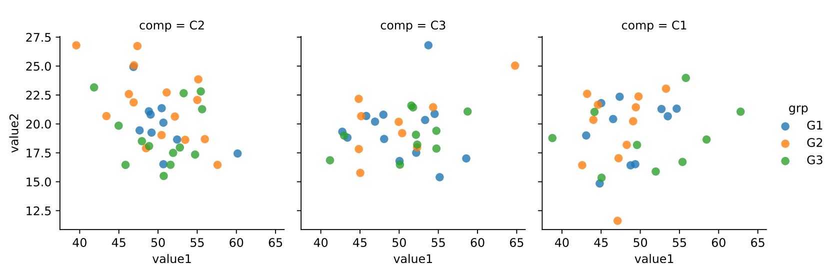
15.3.5 Customization

15.3.5.1 size

size: **height** in inch for each facet

```
sns.lmplot(x='value1', y='value2', col='comp',hue='grp', size=3,fit_reg=False, data=mydf)
```

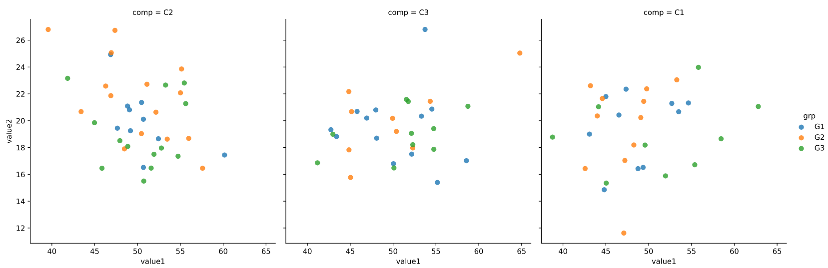
```
plt.show()
```



Observe that even **size is very large**, lmplot will **fit (shrink) everything into one row** by deafult. See example below.

```
sns.lmplot(x='value1', y='value2', col='comp',hue='grp', size=5,fit_reg=False, data=myd
```

```
plt.show()
```

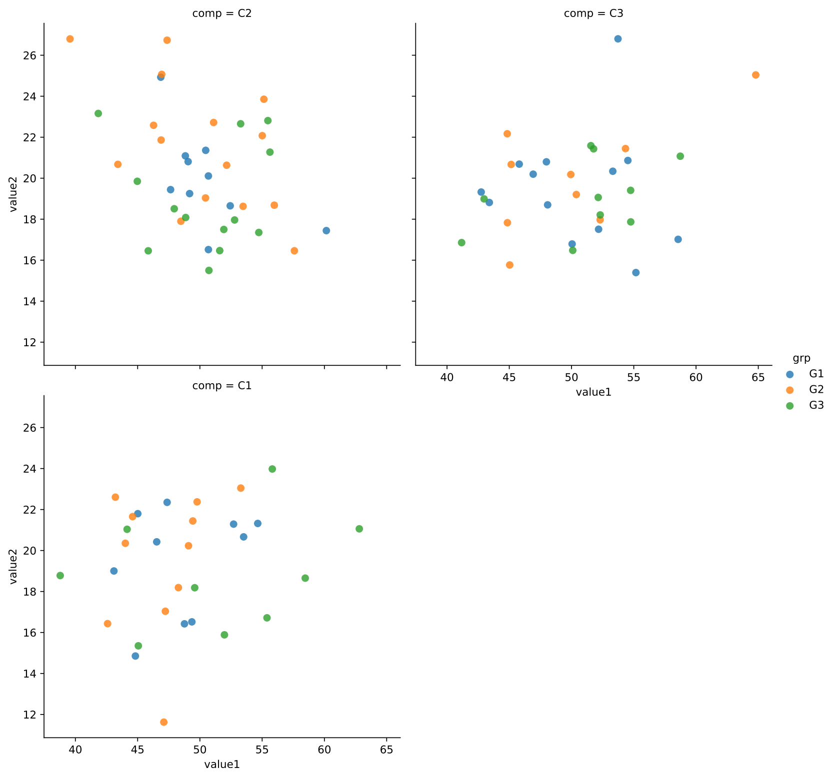


15.3.5.2 col_wrap

To avoid lmplot from shrinking the chart, we use **col_wrap=<col_number** to wrap the output.
Compare the size (height of each facet) with the above **without** col_wrap. Below chart is larger.

```
sns.lmplot(x='value1', y='value2', col='comp',hue='grp', size=5, col_wrap=2, fit_reg=Fa
```

```
plt.show()
```

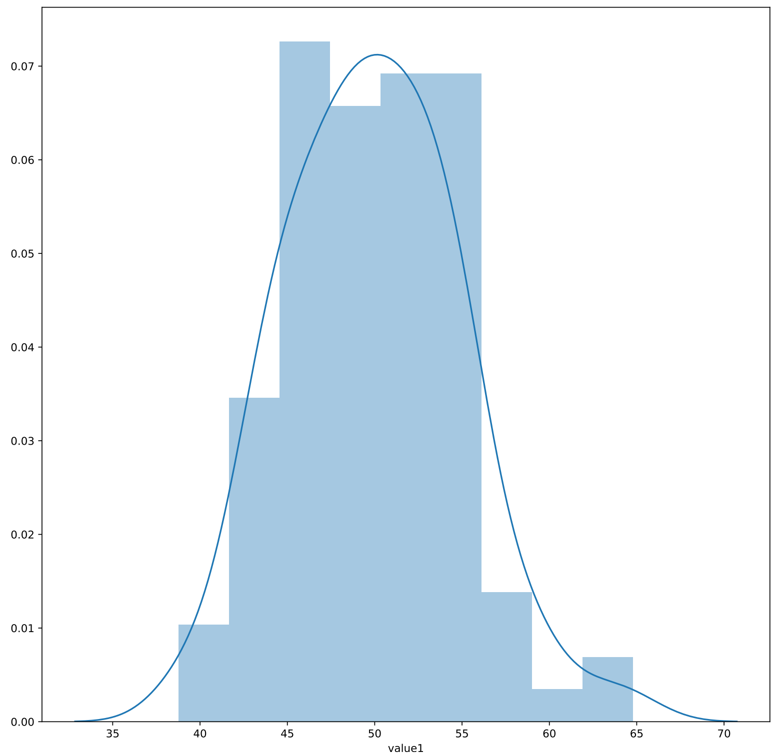


15.4 Histogram

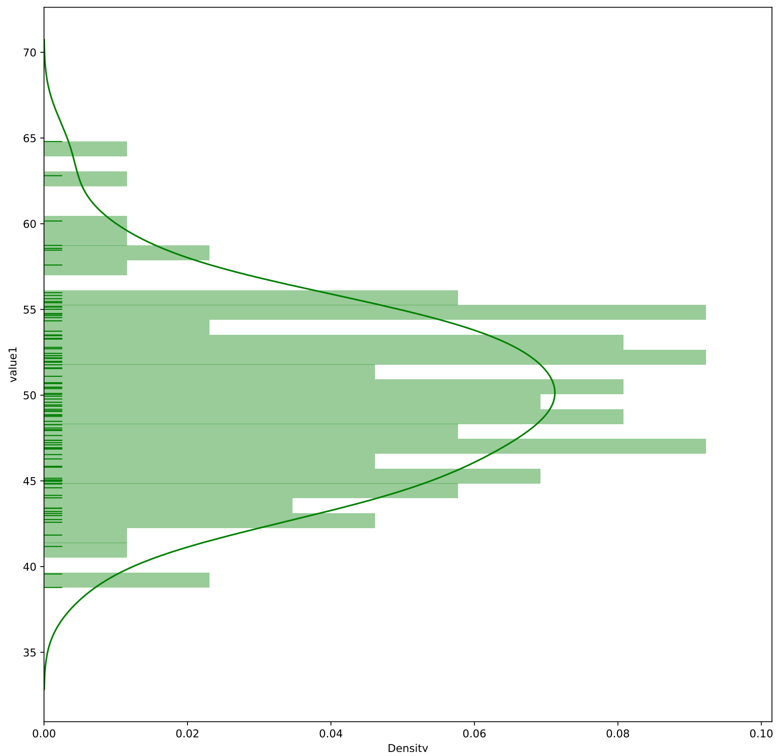
```
seaborn.distplot(
  a,                # Series, 1D Array or List
  bins=None,
  hist=True,
  rug = False,
  vertical=False
)
```

15.4.1 1x Numeric

```
sns.distplot(mydf.value1)
plt.show()
```




```
sns.distplot(mydf.value1,hist=True,rug=True,vertical=True, bins=30,color='g')
plt.show()
```



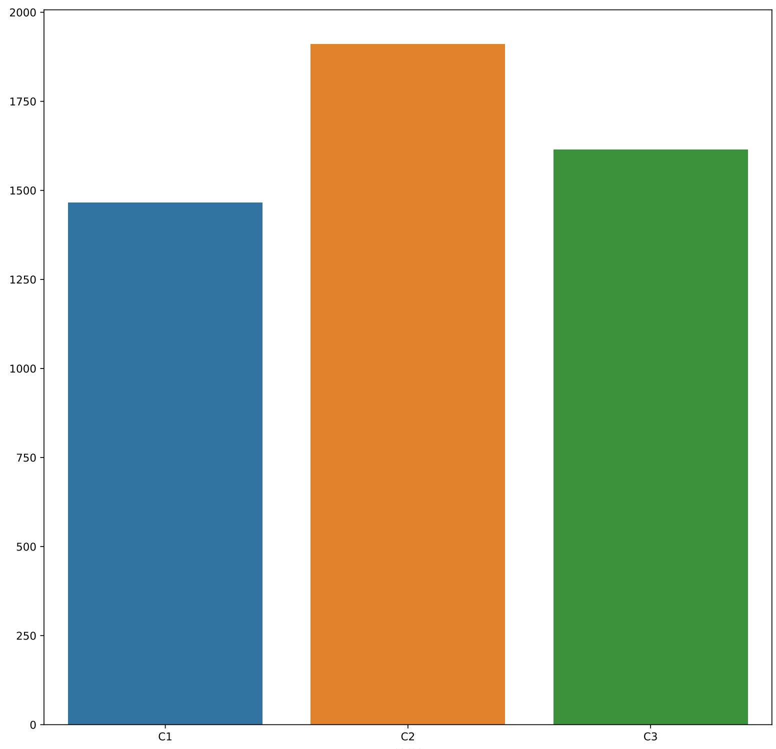
15.5 Bar Chart

```
com_grp = mydf.groupby('comp')
grpdf = com_grp['value1'].sum().reset_index()
grpdf
```

```
#:>   comp       value1
#:> 0   C1  1466.413226
#:> 1   C2  1911.384927
#:> 2   C3  1614.352408
```

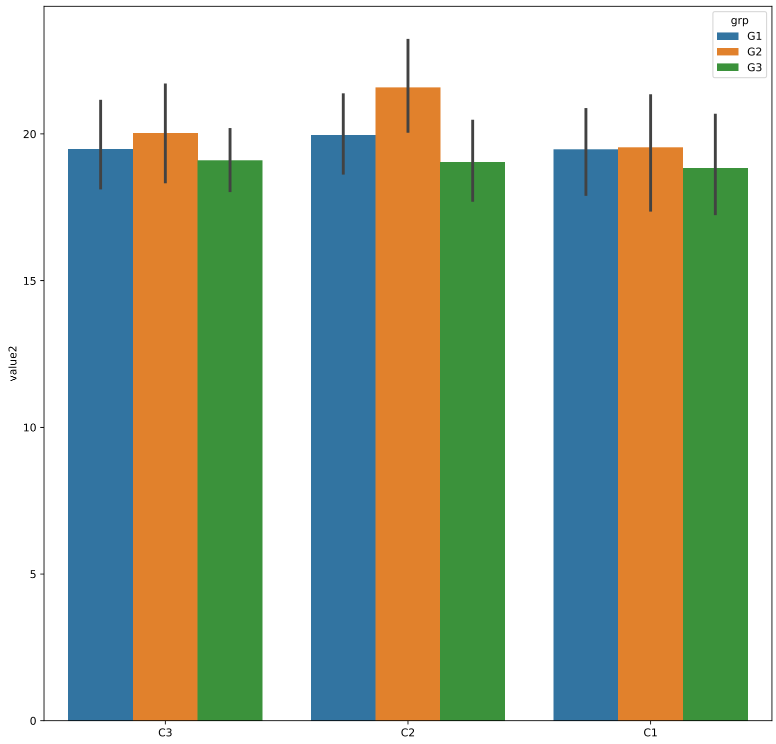
15.5.1 1x Categorical, 1x Numeric

```
sns.barplot(x='comp',y='value1',data=grpdf)
plt.show()
```



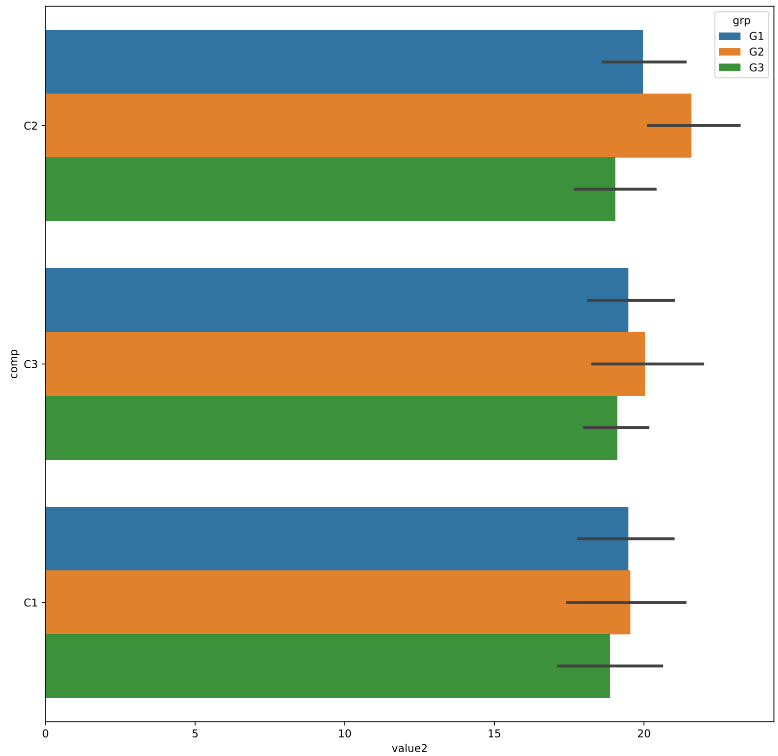
15.5.2 Customization

```
sns.barplot(x='comp',y='value2', hue='grp',
            order=['C3','C2','C1'],
            hue_order=['G1','G2','G3'],
            data=mydf
)
plt.show()
```




```
sns.barplot(x='value2',y='comp', hue='grp',data=mydf)
plt.show()
```

15.5.2.2 Flipping X/Y Axis



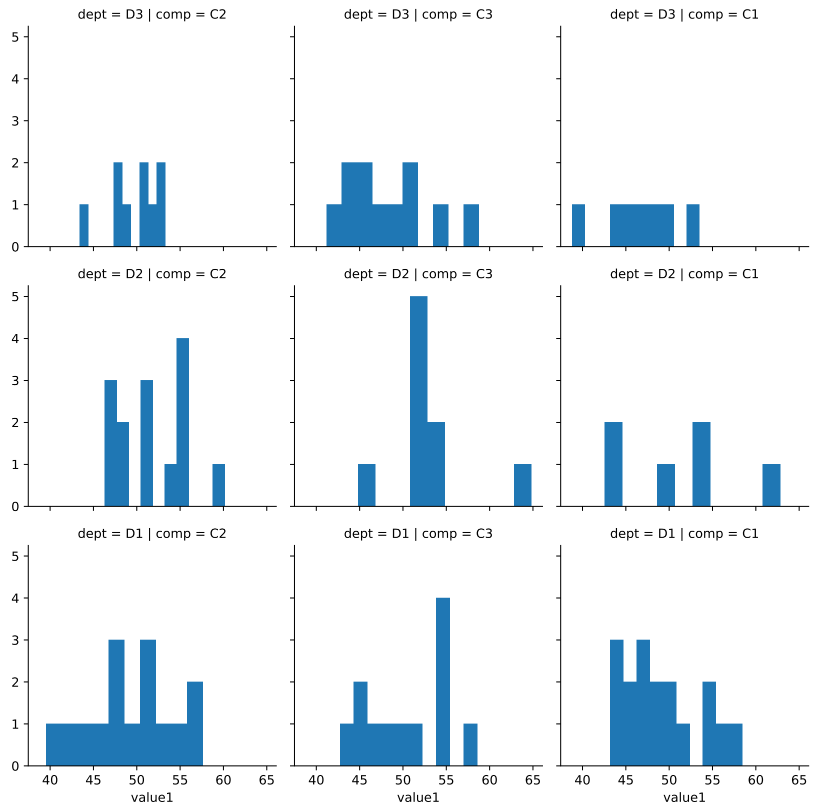
15.6 Faceting

Faceting in Seaborn is a generic function that works with matplotlib various plot utility.
It support matplotlib as well as seaborn plotting utility.

15.6.1 Faceting Hist

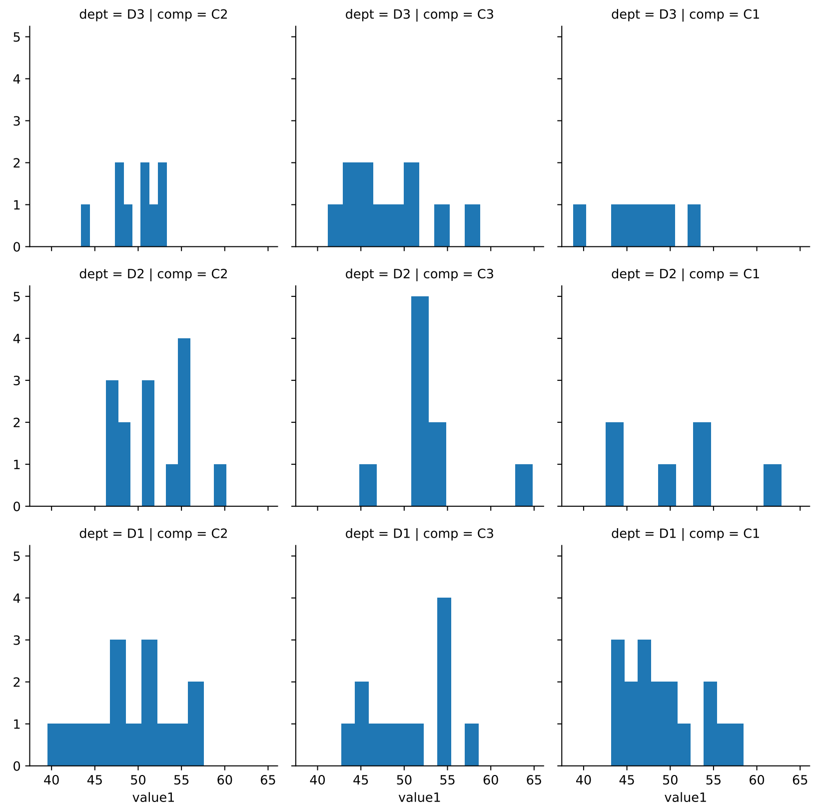
```
g = sns.FacetGrid(mydf, col="comp", row='dept')
g.map(plt.hist, "value1")
```

```
plt.show()
```



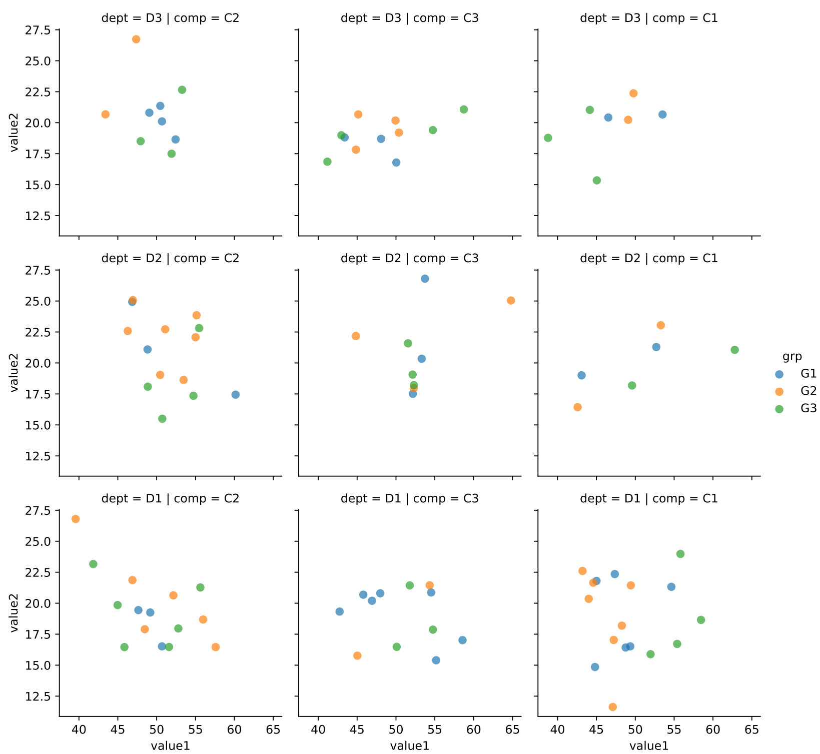

```
g = sns.FacetGrid(mydf, col="comp", row='dept')
g.map(plt.hist, "value1")
```

```
plt.show()
```




```
g = sns.FacetGrid(mydf, col="comp", row='dept',hue='grp')
g.map(plt.scatter, "value1","value2",alpha=0.7);
g.add_legend()
```

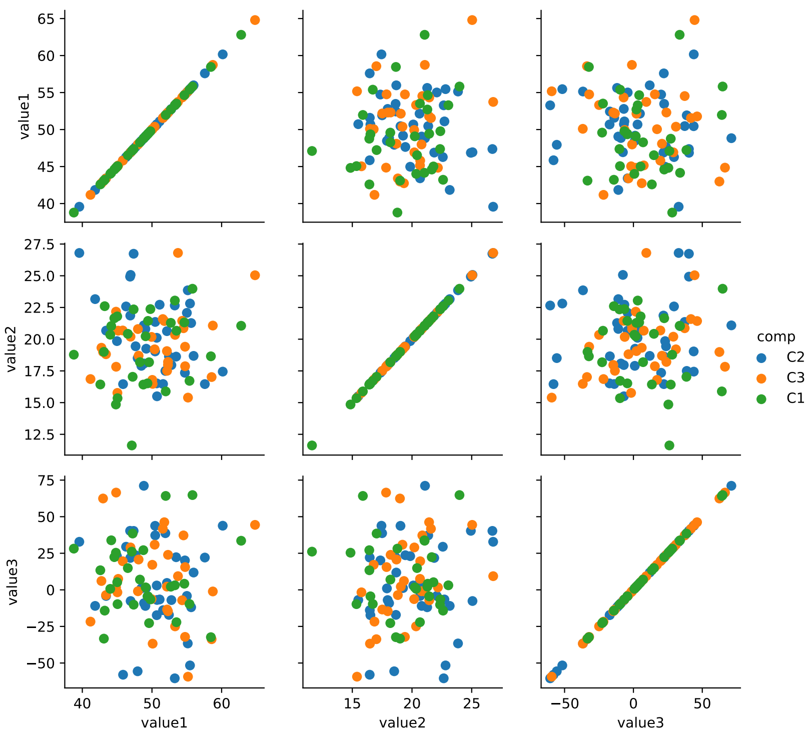
```
plt.show()
```



15.7 Pair Grid

```
g = sns.PairGrid(mydf, hue='comp')
g.map(plt.scatter);
g.add_legend()
```

```
plt.show()
```



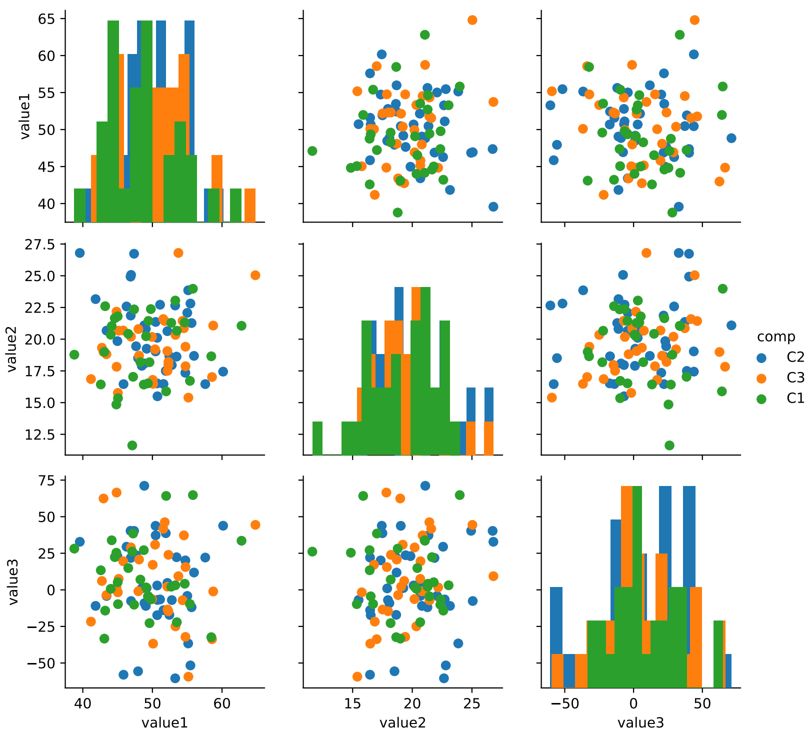

```
g = sns.PairGrid(mydf, hue='comp')
g.map_diag(plt.hist, bins=15)
```

15.7.2 Different Diag and OffDiag

```
g.map_offdiag(plt.scatter)
```

```
g.add_legend()
```

```
plt.show()
```


value1
value2
value3
comp
C2
C3
C1
40
50
60
15
20
25
−50
0
50
65
60
55
50
45
40
27.5
25.0
22.5
20.0
17.5
15.0
12.5
75
50
25
0
−25
−50

Chapter 16

sklearn

This is a machine learning library.

16.1 Setup (hidden)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
pd.set_option( 'display.notebook_repr_html', False)  # render Series and DataFrame as text, not
pd.set_option( 'display.max_column', 10)    # number of columns
pd.set_option( 'display.max_rows', 10)     # number of rows
pd.set_option( 'display.width', 90)        # number of characters per row
```

16.2 The Library

sklearn **does not automatically import its subpackages**. Therefore all subpakcages must be specifically loaded before use.

```
# Sample Data
from sklearn                 import datasets

# Model Selection
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import cross_validate

# Preprocessing
```

```
# from sklearn.preprocessing   import Imputer
from sklearn.impute import SimpleImputer

from sklearn.preprocessing   import MinMaxScaler
from sklearn.preprocessing   import StandardScaler
from sklearn.preprocessing   import Normalizer
from sklearn.preprocessing   import PolynomialFeatures

# Model and Pipeline
from sklearn.linear_model    import LinearRegression,Lasso
from sklearn.pipeline        import make_pipeline

# Measurement
from sklearn.metrics         import *

import statsmodels.formula.api as smf
```

16.3 Model Fitting

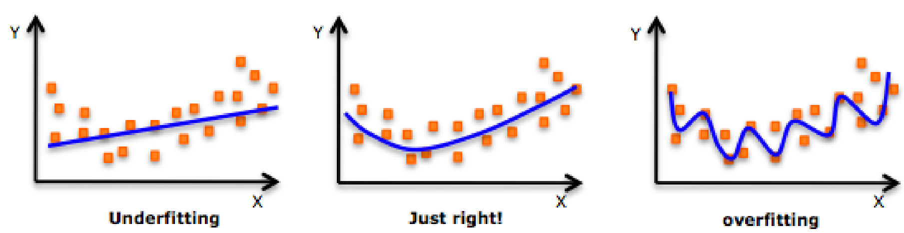


Figure 16.1: split

16.3.1 Underfitting

- The model does not fit the training data and therefore misses the trends in the data
- The model cannot be generalized to new data, this is usually the result of a very simple model (not enough predictors/independent variables)
- The model will have poor predictive ability
- For example, we fit a linear model (like linear regression) to data that is not linear

16.3.2 Overfitting

- The model has trained ?too well? and is now, well, fit too closely to the training dataset

- The model is too complex (i.e. **too many features/variables** compared to the number of observations)
- The model will be very accurate on the training data but will probably be very not accurate on untrained or new data
- The model is not generalized (or not AS generalized), meaning you can generalize the results
- The model learns or describes the ?noise? in the training data instead of the actual relationships between variables in the data

16.3.3 Just Right

- It is worth noting the underfitting is not as prevalent as overfitting
- Nevertheless, we want to avoid both of those problems in data analysis
- We want to find the middle ground between under and overfitting our model

16.4 Model Tuning

- A highly complex model tend to overfit
- A too flexible model tend to underfit

Complexity can be reduced by: - Less features - Less degree of polynomial features - Apply generalization (tuning hyperparameters)

16.5 High Level ML Process

16.6 Built-in Datasets

sklearn included some popular datasets to play with
Each dataset is of type **Bunch**.
It has useful data (array) in the form of properties:
- keys (display all data availabe within the dataset)
- data (common)
- target (common)
- DESCR (common) - feature_names (some dataset)
- target_names (some dataset) - images (some dataset)

16.6.1 diabetes (regression)

16.6.1.1 Load Dataset

```
diabetes = datasets.load_diabetes()
print (type(diabetes))
```

```
#:> <class 'sklearn.utils.Bunch'>
```

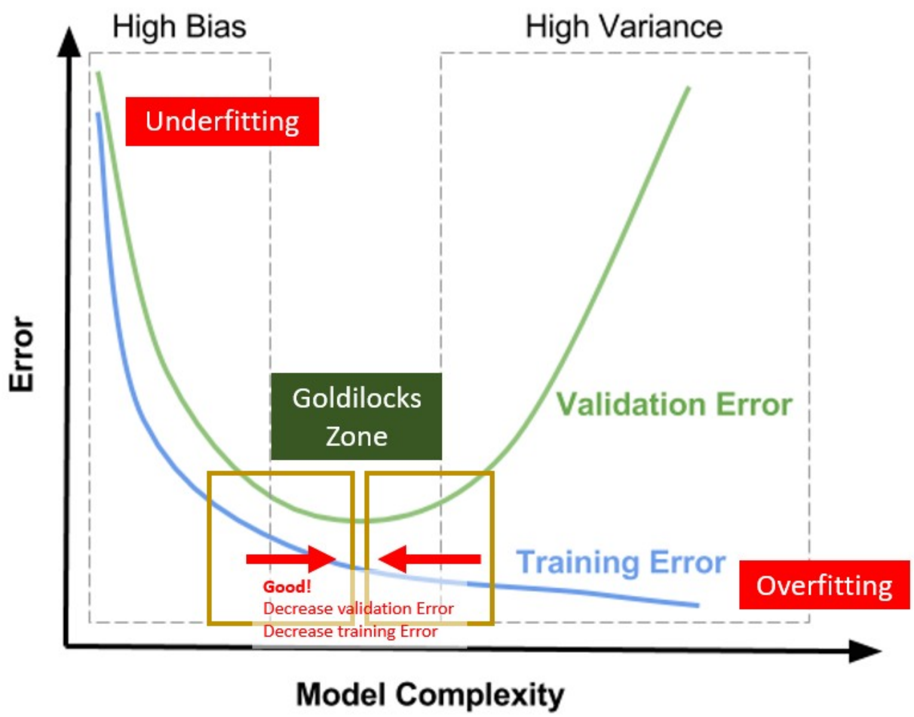


Figure 16.2: split

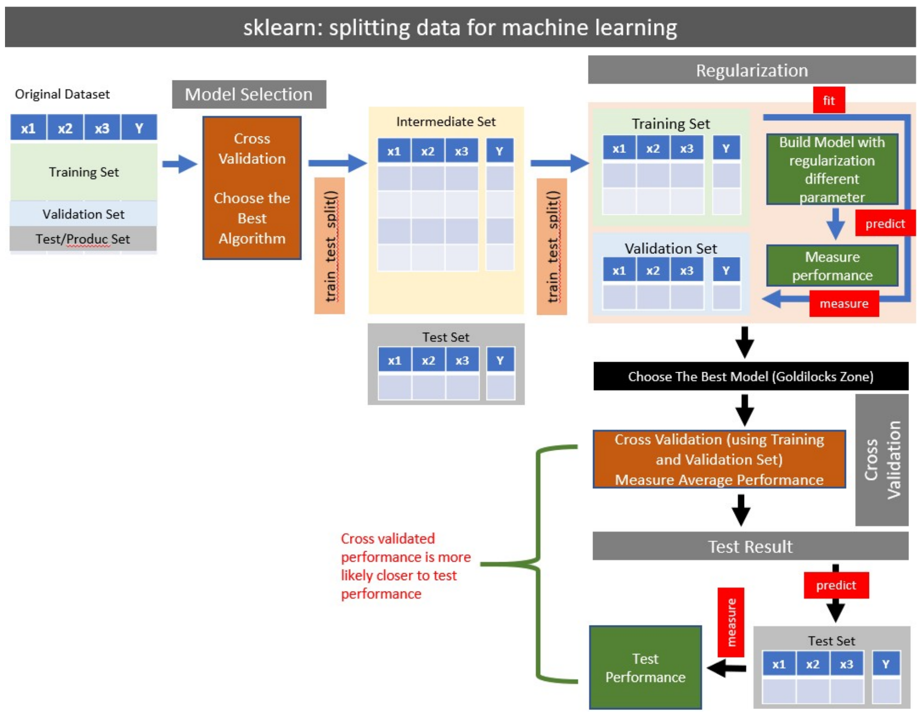


Figure 16.3: split

16.6.1.2 keys

```
diabetes.keys()
```

```
#:> dict_keys(['data', 'target', 'frame', 'DESCR', 'feature_names', 'data_filename', '
```

16.6.1.3 Features and Target

.data = features - two dimension array
.target = target - one dimension array

```
print (type(diabetes.data))
```

```
#:> <class 'numpy.ndarray'>
```

```
print (type(diabetes.target))
```

```
#:> <class 'numpy.ndarray'>
```

```
print (diabetes.data.shape)
```

```
#:> (442, 10)
```

```
print (diabetes.target.shape)
```

```
#:> (442,)
```

16.6.1.4 Load with X,y (Convenient Method)

using return_X_y = True, data is loaded into X, target is loaded into y

```
X,y      = datasets.load_diabetes(return_X_y=True)
```

```
print (X.shape)
```

```
#:> (442, 10)
```

```
print (y.shape)
```

```
#:> (442,)
```

16.6.2 digits (Classification)

This is a copy of the test set of the UCI ML hand-written digits datasets

```
digits = datasets.load_digits()
print (type(digits))
```

```
#:> <class 'sklearn.utils.Bunch'>
```

```
print (type(digits.data))
```

```
#:> <class 'numpy.ndarray'>
```

```
digits.keys()
```

```
#:> dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

```
digits.target_names
```

```
#:> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

16.6.2.1 data

```
digits.data.shape  # features
```

```
#:> (1797, 64)
```

```
digits.target.shape # target
```

```
#:> (1797,)
```

16.6.2.2 Images

- images is 3 dimensional array
- There are **1797 samples**, each sample is **8x8 pixels**

```
digits.images.shape
```

```
#:> (1797, 8, 8)
```

```
type(digits.images)
```

```
#:> <class 'numpy.ndarray'>
```

Each element represent the data that make its target

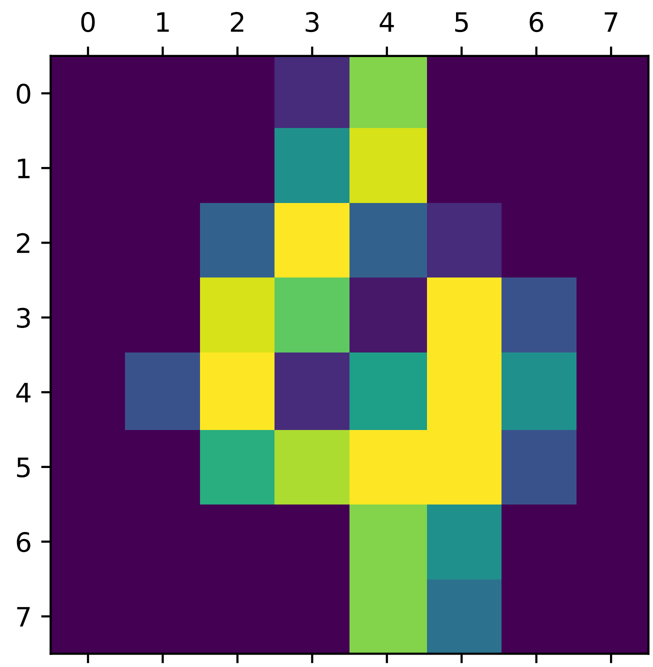
```
print (digits.target[100])
```

```
#:> 4
```

```
print (digits.images[100])
```

```
#:> [[ 0.  0.  0.  2. 13.  0.  0.  0.]
#:>  [ 0.  0.  0.  8. 15.  0.  0.  0.]
#:>  [ 0.  0.  5. 16.  5.  2.  0.  0.]
#:>  [ 0.  0. 15. 12.  1. 16.  4.  0.]
#:>  [ 0.  4. 16.  2.  9. 16.  8.  0.]
#:>  [ 0.  0. 10. 14. 16. 16.  4.  0.]
#:>  [ 0.  0.  0.  0. 13.  8.  0.  0.]
#:>  [ 0.  0.  0.  0. 13.  6.  0.  0.]]
```

```
plt.matshow(digits.images[100])
```



16.6.2.3 Loading Into X,y (Convenient Method)

```
X,y = datasets.load_digits(return_X_y=True)
```

```
X.shape
```

```
#:> (1797, 64)
```

```
y.shape
```

```
#:> (1797,)
```

16.6.3 iris (Classification)

```
iris = datasets.load_iris()
```

```
iris.keys()
```

```
#:> dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'f
```

16.6.3.1 Feature Names

```
iris.feature_names
```

```
#:> ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

16.6.3.2 target

```
iris.target_names
```

```
#:> array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

```
iris.target
```

```
#:> array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
#:>        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
#:>        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
#:>        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
#:>        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
#:>        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
#:>        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

16.7 Train Test Data Splitting

16.7.1 Sample Data

Generate 100 rows of data, with 3x features (X1,X2,X3), and one dependant variable (Y)

```
n = 21  # number of samples
I = 5  # intercept value
E = np.random.randint( 1,20, n)  # Error
x1 = np.random.randint( 1,n+1, n)
x2 = np.random.randint( 1,n+1, n)
x3 = np.random.randint( 1,n+1, n)
y = 0.1*x1 + 0.2*x2 + 0.3*x3 + E + I
mydf = pd.DataFrame({
    'y':y,
    'x1':x1,
    'x2':x2,
    'x3':x3
})
mydf.shape
```

```
#:> (21, 4)
```

16.7.2 One Time Split

sklearn::train_test_split() has two forms: - Take one DF, split into 2 DF (most of sklearn modeling use this method - Take two DFs, split into 4 DF

```
mydf.head()
```

```
#:>       y  x1  x2  x3
#:> 0  14.0   3  21   5
#:> 1   9.3   4   1   9
#:> 2  19.3  12  17  19
#:> 3  13.8  19   9   7
#:> 4  24.9  20  19  17
```

16.7.2.1 Method 1: Split One Dataframe Into Two (Train & Test)

```
traindf, testdf = train_test_split( df, test_size=, random_state= )
 # random_state : seed number (integer), optional
 # test_size    : fraction of 1, 0.2 means 20%
```

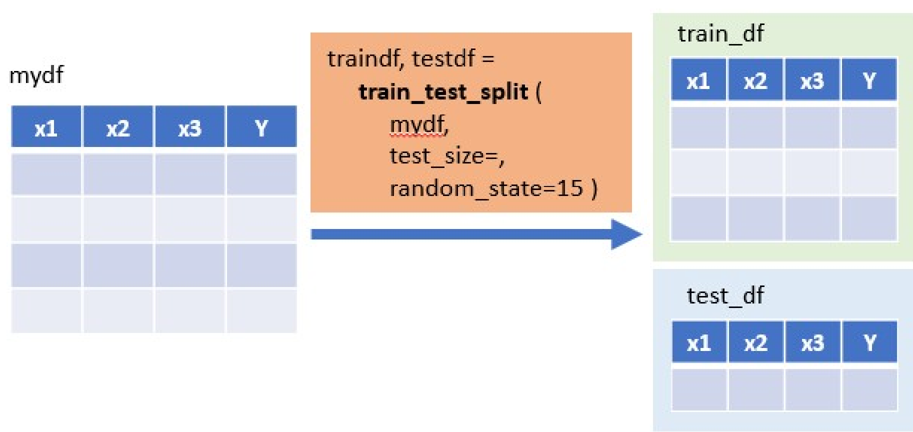


Figure 16.4: split

```
traindf, testdf = train_test_split(mydf,test_size=0.2, random_state=25)
```

```
print (len(traindf))
```

```
#:> 16
```

```
print (len(testdf))
```

```
#:> 5
```

16.7.2.2 Method 2: Split Two DataFrame (X,Y) into Four x_train/test, y_train/test

```
x_train, x_test, y_train, y_test = train_test_split( X,Y, test_size=, random_state= )
 # random_state : seed number (integer), optional
 # test_size    : fraction of 1, 0.2 means 20%
```

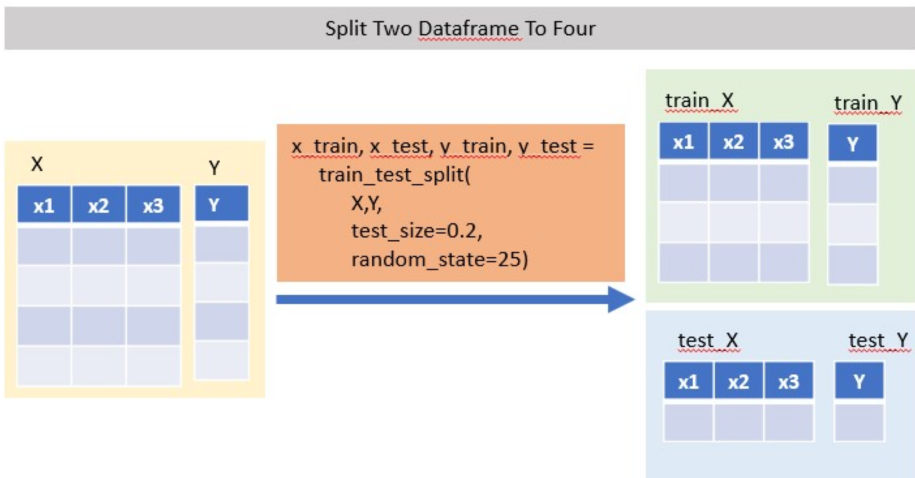


Figure 16.5: split

Split DataFrame into X and Y First

```
feature_cols = ['x1','x2','x3']
X = mydf[feature_cols]
Y = mydf.y
```

Then Split X/Y into x_train/test, y_train/test

```
x_train, x_test, y_train, y_test = train_test_split( X,Y, test_size=0.2, random_state=25)
print (len(x_train))
```

```
#:> 16
```

```
print (len(x_test))
```

```
#:> 5
```

16.7.3 K-Fold

```
KFold(n_splits=3, shuffle=False, random_state=None)
```

suffle=False (default), meaning index number is taken continously

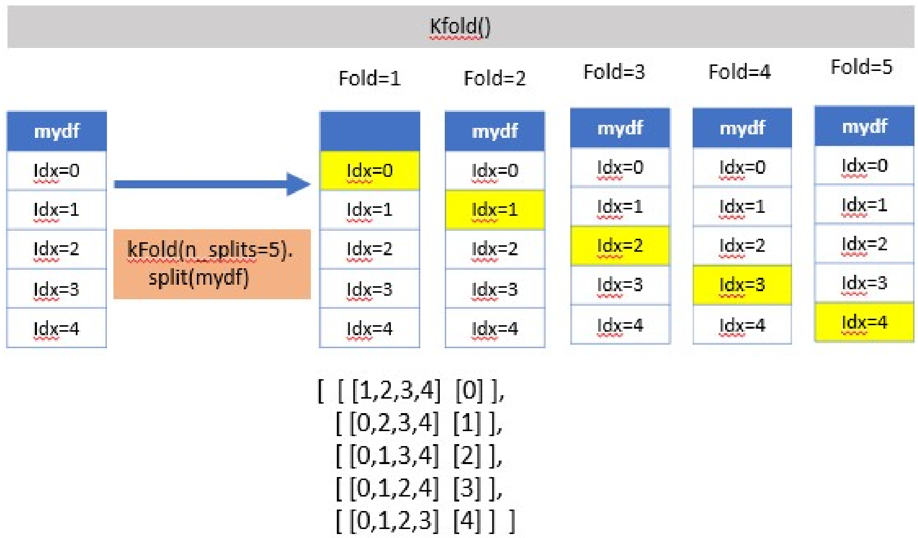


Figure 16.6: split

```
kf = KFold(n_splits=7)
```

```
for train_index, test_index in kf.split(X):
  print (train_index, test_index)
```

```
#:> [ 3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [0 1 2]
#:> [ 0  1  2  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [3 4 5]
#:> [ 0  1  2  3  4  5  9 10 11 12 13 14 15 16 17 18 19 20] [6 7 8]
#:> [ 0  1  2  3  4  5  6  7  8 12 13 14 15 16 17 18 19 20] [ 9 10 11]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 15 16 17 18 19 20] [12 13 14]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 18 19 20] [15 16 17]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17] [18 19 20]
```

shuffle=True

```
kf = KFold(n_splits=7, shuffle=True)
```

```
for train_index, test_index in kf.split(X):
  print (train_index, test_index)
```

```
#:> [ 0  1  3  4  5  6  7  8  9 10 11 12 14 15 17 18 19 20] [ 2 13 16]
#:> [ 0  1  2  3  4  5  8  9 10 11 12 13 14 15 16 18 19 20] [ 6  7 17]
#:> [ 0  1  2  3  5  6  7  8  9 11 12 13 14 15 16 17 18 19] [ 4 10 20]
#:> [ 0  1  2  4  5  6  7  8 10 11 12 13 15 16 17 18 19 20] [ 3  9 14]
#:> [ 1  2  3  4  5  6  7  9 10 11 12 13 14 16 17 18 19 20] [ 0  8 15]
#:> [ 0  2  3  4  6  7  8  9 10 11 13 14 15 16 17 18 19 20] [ 1  5 12]
```

```
#:> [ 0  1  2  3  4  5  6  7  8  9 10 12 13 14 15 16 17 20] [11 18 19]
```

16.7.4 Leave One Out

- For a dataset of N rows, Leave One Out will split N-1 times, each time leaving one row as test, remaning as training set.

- Due to the **high number of test sets** (which is the same as the number of samples-1) this cross-validation method can be very costly. For large datasets one should favor KFold.

```
loo = LeaveOneOut()
```

```
for train_index, test_index in loo.split(X):
  print (train_index, test_index)
```

```
#:> [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [0]
#:> [ 0  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [1]
#:> [ 0  1  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [2]
#:> [ 0  1  2  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [3]
#:> [ 0  1  2  3  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [4]
#:> [ 0  1  2  3  4  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [5]
#:> [ 0  1  2  3  4  5  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [6]
#:> [ 0  1  2  3  4  5  6  8  9 10 11 12 13 14 15 16 17 18 19 20] [7]
#:> [ 0  1  2  3  4  5  6  7  9 10 11 12 13 14 15 16 17 18 19 20] [8]
#:> [ 0  1  2  3  4  5  6  7  8 10 11 12 13 14 15 16 17 18 19 20] [9]
#:> [ 0  1  2  3  4  5  6  7  8  9 11 12 13 14 15 16 17 18 19 20] [10]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 12 13 14 15 16 17 18 19 20] [11]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 13 14 15 16 17 18 19 20] [12]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 14 15 16 17 18 19 20] [13]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 15 16 17 18 19 20] [14]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 16 17 18 19 20] [15]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 17 18 19 20] [16]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 18 19 20] [17]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 19 20] [18]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 20] [19]
#:> [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19] [20]
```

```
X
```

```
#:>     x1  x2  x3
#:> 0    3  21   5
#:> 1    4   1   9
#:> 2   12  17  19
#:> 3   19   9   7
#:> 4   20  19  17
#:> ..  ..  ..  ..
#:> 16  16  10   4
```

```
#:> 17   6  17  15
#:> 18   1   4   4
#:> 19  10   5  10
#:> 20  17  12   8
#:>
#:> [21 rows x 3 columns]
```

16.8 Polynomial Transform

This can be used as part of feature engineering, to introduce new features for data that seems to fit with quadradic model.

16.8.1 Single Variable

16.8.1.1 Sample Data

Data must be 2-D before polynomial features can be applied. Code below convert 1D array into 2D array.

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

```
#:> array([[1],
#:>        [2],
#:>        [3],
#:>        [4],
#:>        [5]])
```

16.8.1.2 Degree 1

One Degree means maintain original features. No new features is created.

```
PolynomialFeatures(degree=1, include_bias=False).fit_transform(X)
```

```
#:> array([[1.],
#:>        [2.],
#:>        [3.],
#:>        [4.],
#:>        [5.]])
```

16.8.1.3 Degree 2

Degree-1 original feature: x
Degree-2 additional features: x^2

```
PolynomialFeatures(degree=2, include_bias=False).fit_transform(X)
```

```
#:> array([[ 1.,  1.],
#:>        [ 2.,  4.],
#:>        [ 3.,  9.],
#:>        [ 4., 16.],
#:>        [ 5., 25.]])
```

16.8.1.4 Degree 3

Degree-1 original feature: x
Degree-2 additional features: x^2
Degree-3 additional features: x^3

```
PolynomialFeatures(degree=3, include_bias=False).fit_transform(X)
```

```
#:> array([[  1.,   1.,   1.],
#:>        [  2.,   4.,   8.],
#:>        [  3.,   9.,  27.],
#:>        [  4.,  16.,  64.],
#:>        [  5.,  25., 125.]])
```

16.8.1.5 Degree 4

Degree-1 original feature: x
Degree-2 additional features: x^2
Degree-3 additional features: x^3
Degree-3 additional features: x^4

```
PolynomialFeatures(degree=4, include_bias=False).fit_transform(X)
```

```
#:> array([[  1.,   1.,   1.,   1.],
#:>        [  2.,   4.,   8.,  16.],
#:>        [  3.,   9.,  27.,  81.],
#:>        [  4.,  16.,  64., 256.],
#:>        [  5.,  25., 125., 625.]])
```

16.8.2 Two Variables

16.8.2.1 Sample Data

```
X = pd.DataFrame( {'x1': [1, 2, 3, 4, 5 ],
                   'x2': [6, 7, 8, 9, 10]})
X
```

```
#:>    x1  x2
#:> 0   1   6
#:> 1   2   7
#:> 2   3   8
#:> 3   4   9
```

```
#:> 4   5  10
```

16.8.2.2 Degree 2

```
Degree-1 original   features:  x1,     x2
Degree-2 additional features:  x1^2,   x2^2,   x1:x2
```

```
PolynomialFeatures(degree=2, include_bias=False).fit_transform(X)
```

```
#:> array([[  1.,   6.,   1.,   6.,  36.],
#:>        [  2.,   7.,   4.,  14.,  49.],
#:>        [  3.,   8.,   9.,  24.,  64.],
#:>        [  4.,   9.,  16.,  36.,  81.],
#:>        [  5.,  10.,  25.,  50., 100.]])
```

16.8.2.3 Degree 3

```
Degree-1 original   features:  x1,       x2
Degree-2 additional features:  x1^2,     x2^2,   x1:x2
Degree-3 additional features:  x1^3,     x2^3    x1:x2^2    x2:x1^2
```

```
PolynomialFeatures(degree=3, include_bias=False).fit_transform(X)
```

```
#:> array([[   1.,    6.,    1.,    6.,   36.,    1.,    6.,   36.,  216.],
#:>        [   2.,    7.,    4.,   14.,   49.,    8.,   28.,   98.,  343.],
#:>        [   3.,    8.,    9.,   24.,   64.,   27.,   72.,  192.,  512.],
#:>        [   4.,    9.,   16.,   36.,   81.,   64.,  144.,  324.,  729.],
#:>        [   5.,   10.,   25.,   50.,  100.,  125.,  250.,  500., 1000.]])
```

16.9 Imputation of Missing Data

16.9.1 Sample Data

```
from numpy import nan
X = np.array([[ nan, 0,   3  ],
              [ 3,   7,   9  ],
              [ 3,   5,   2  ],
              [ 4,   nan, 6  ],
              [ 8,   8,   1  ]])

y = np.array([14, 16, -1,  8, -5])
```

16.9.2 Imputer

16.9.2.1 mean strategy

```
imp = SimpleImputer(strategy='mean')
X2 = imp.fit_transform(X)
X2
```

```
#:> array([[4.5, 0. , 3. ],
#:>        [3. , 7. , 9. ],
#:>        [3. , 5. , 2. ],
#:>        [4. , 5. , 6. ],
#:>        [8. , 8. , 1. ]])
```

16.10 Scaling

It is possible that some insignificant variable with larger range will be dominating the objective function.
We can remove this problem by scaling down all the features to a same range.

16.10.1 Sample Data

```
X=mydf.filter(like='x')[:5]
X
```

```
#:>    x1  x2  x3
#:> 0   3  21   5
#:> 1   4   1   9
#:> 2  12  17  19
#:> 3  19   9   7
#:> 4  20  19  17
```

16.10.2 MinMax Scaler

```
MinMaxScaler( feature_range(0,1), copy=True )
# default feature range (output result) from 0 to 1
# default return a copy of new array, copy=False will inplace original array
```

Define Scaler Object

```
scaler = MinMaxScaler()
```

Transform Data

```
scaler.fit_transform(X)
```

```
#:> array([[0.        , 1.        , 0.        ],
#:>        [0.05882353, 0.        , 0.28571429],
```

```
#:>        [0.52941176, 0.8       , 1.        ],
#:>        [0.94117647, 0.4       , 0.14285714],
#:>        [1.        , 0.9       , 0.85714286]])
```

Scaler Attributes

```
data_min_: minimum value of the feature (before scaling)
data_max_: maximum value of the feature (before scaling)
```

```
pd.DataFrame(list(zip(scaler.data_min_, scaler.data_max_)),
             columns=['data_min','data_max'],
             index=X.columns)
```

```
#:>     data_min  data_max
#:> x1       3.0      20.0
#:> x2       1.0      21.0
#:> x3       5.0      19.0
```

16.10.3 Standard Scaler

It is most suitable for techniques that assume a Gaussian distribution in the input variables and work better with rescaled data, such as linear regression, logistic regression and linear discriminate analysis.

```
StandardScaler(copy=True, with_mean=True, with_std=True)
# copy=True : return a copy of data, instead of inplace
# with_mean=True : centre all features by substracting with its mean
# with_std=True  : centre all features by dividing with its std
```

Define Scaler Object

```
scaler = StandardScaler()
```

Transform Data

```
scaler.fit_transform(X)
```

```
#:> array([[-1.19907948,  1.02441135, -1.14873305],
#:>        [-1.05965163, -1.67140798, -0.4307749 ],
#:>        [ 0.05577114,  0.48524748,  1.3641205 ],
#:>        [ 1.03176607, -0.59308025, -0.78975397],
#:>        [ 1.17119391,  0.75482941,  1.00514142]])
```

Scaler Attributes
After the data transformation step above, scaler will have the mean and variance information for each feature.

```
pd.DataFrame(list(zip(scaler.mean_, scaler.var_)),
             columns=['mean','variance'],
             index=X.columns)
```

```
#:>     mean  variance
#:> x1  11.6     51.44
#:> x2  13.4     55.04
#:> x3  11.4     31.04
```

16.11 Pipeline

With any of the preceding examples, it can quickly become tedious to do the transformations by hand, especially if you wish to string together multiple steps. For example, we might want a processing pipeline that looks something like this:

- **Impute** missing values using the mean
- **Transform** features to quadratic
- **Fit** a linear regression

make_pipeline takes list of functions as parameters. When calling **fit()** on a pipeline object, these functions will be performed in sequential with data flow from one function to another.

```
make_pipeline (
    function_1 (),
    function_2 (),
    function_3 ()
 )
```

16.11.1 Sample Data

```
X
```

```
#:>    x1  x2  x3
#:> 0   3  21   5
#:> 1   4   1   9
#:> 2  12  17  19
#:> 3  19   9   7
#:> 4  20  19  17
```

```
y
```

```
#:> array([14, 16, -1,  8, -5])
```

16.11.2 Create Pipeline

```
my_pipe = make_pipeline (
    SimpleImputer            (strategy='mean'),
    PolynomialFeatures (degree=2),
```

```
    LinearRegression   ()
)
type(my_pipe)
```

```
#:> <class 'sklearn.pipeline.Pipeline'>
```

```
my_pipe
```

```
#:> Pipeline(steps=[('simpleimputer', SimpleImputer()),
#:>                 ('polynomialfeatures', PolynomialFeatures()),
#:>                 ('linearregression', LinearRegression())])
```

16.11.3 Executing Pipeline

```
my_pipe.fit( X, y) # execute the pipeline
```

```
#:> Pipeline(steps=[('simpleimputer', SimpleImputer()),
#:>                 ('polynomialfeatures', PolynomialFeatures()),
#:>                 ('linearregression', LinearRegression())])
```

```
print (y)
```

```
#:> [14 16 -1  8 -5]
```

```
print (my_pipe.predict(X))
```

```
#:> [14. 16. -1.  8. -5.]
```

```
type(my_pipe)
```

```
#:> <class 'sklearn.pipeline.Pipeline'>
```

16.12 Cross Validation

16.12.1 Load Data

```
X,y = datasets.load_diabetes(return_X_y=True)
```

16.12.2 Choose An Cross Validator

```
kf = KFold(n_splits=5)
```

16.12.3 Run Cross Validation

Single Scorer
Use default scorer of the estimator (if available)

```
lasso = Lasso()
cv_results1 = cross_validate(lasso, X,y,cv=kf,
    return_train_score=False)
```

Multiple Scorer
Specify the scorer http://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

```
cv_results2 = cross_validate(lasso, X,y,cv=kf,
    scoring=("neg_mean_absolute_error","neg_mean_squared_error","r2"),
    return_train_score=False)
```

16.12.4 The Result

Result is a **dictionary**

```
cv_results1.keys()
```

```
#:> dict_keys(['fit_time', 'score_time', 'test_score'])
```

```
cv_results2.keys()
```

```
#:> dict_keys(['fit_time', 'score_time', 'test_neg_mean_absolute_error', 'test_neg_mean_squared_e
```

```
cv_results1
```

```
#:> {'fit_time': array([0.00081229, 0.00317311, 0.00122571, 0.0007627 , 0.00070167]), 'score_time
```

```
cv_results2
```

```
#:> {'fit_time': array([0.00092649, 0.00089145, 0.00072384, 0.00085139, 0.0013237 ]), 'score_time
50.09003423, -52.54110842, -55.02813846, -50.81121806,
#:>        -55.60471593]), 'test_neg_mean_squared_error': array([-
3491.74009759, -4113.86002091, -4046.91780932, -3489.74018715,
#:>        -4111.92401769]), 'test_r2': array([0.28349047, 0.35157959, 0.3533813 , 0.33481474, 0.
```

Chapter 17

NLP

Natural Language Processing

17.1 Regular Expression

- Rgular expressions (called REs or regexes) is mandatory skill for NLP. The `re` is a **built-in* library

- It is essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module

- Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C

17.1.1 Syntax

There are two methods to emply re. Below method compile a regex first, then apply it multiple times in subsequent code.

```
import re
pattern = re.compile(r'put pattern here')
pattern.match('put text here')
```

Second method below employ compile and match in single line. The pattern cannot be reused, therefore good for onetime usage only.

```
import re
pattern = (r'put pattern here')
re.match(pattern, r'put text here')  # compile and match in single line
```

17.1.2 Finding

17.1.2.1 Find The First Match

There are two ways to find the first match:
- **`re.search`** find first match anywhere in text, including multiline
- **`re.match`** find first match at the BEGINNING of text, similar to `re.search`with ^
- Both returns first match, return **MatchObject**
- Both returns **None** if no match is found

```
pattern1 = re.compile('123')
pattern2 = re.compile('123')
pattern3 = re.compile('^123')  # equivalent to above
text = 'abc123xyz'

## Single Line Text Example
print( 're.search found a match somewhere:\n',
       pattern1.search(text), '\n', ## found
       '\nre.match did not find anything at the beginning:\n',
       pattern2.match(text), '\n',
       '\nre.search did not find anything at beginning too:\n',
       pattern3.search(text))        ## None
```

```
#:> re.search found a match somewhere:
#:>  <re.Match object; span=(3, 6), match='123'>
#:>
#:> re.match did not find anything at the beginning:
#:>  None
#:>
#:> re.search did not find anything at beginning too:
#:>  None
```

Returned **MatchObject** provides useful information about the matched string.

```
age_pattern = re.compile(r'\d+')
age_text    = 'Ali is my teacher. He is 109 years old. his kid is 40 years old.'
first_found = age_pattern.search(age_text)

print('Found Object:           ', first_found,
      '\nInput Text:             ', first_found.string,
      '\nInput Pattern:          ', first_found.re,
      '\nFirst Found string:     ', first_found.group(),
      '\nFound Start Position:   ', first_found.start(),
      '\nFound End Position:     ', first_found.end(),
      '\nFound Span:             ', first_found.span(),)
```

```
#:> Found Object:           <re.Match object; span=(25, 28), match='109'>
```

```
#:> Input Text:             Ali is my teacher. He is 109 years old. his kid is 40 years old.
#:> Input Pattern:          re.compile('\\d+')
#:> First Found string:     109
#:> Found Start Position:   25
#:> Found End Position:     28
#:> Found Span:             (25, 28)
```

17.1.2.2 Find All Matches

`findall()` returns all matching string as **list**. If no matches found, it return an empty list.

```
print(
  'Finding Two Digits:',
  re.findall(r'\d\d','abc123xyz456'), '\n',
  '\nFound Nothing:',
  re.findall(r'\d\d','abcxyz'))
```

```
#:> Finding Two Digits: ['12', '45']
#:>
#:> Found Nothing: []
```

17.1.3 Matching Condition

17.1.3.1 Meta Characters

```
[]     match any single character within the bracket
[1234] is the same as [1-4]
[0-39] is the same as [01239]
[a-e]  is the same as [abcde]
[^abc] means any character except a,b,c
[^0-9] means any character except 0-9
a|b:   a or b
{n,m}  at least n repetition, but maximum m repetition
()     grouping
```

```
pattern = re.compile(r'[a-z]+')
text1 = "tempo"
text2 = "tempo1"
text3 = "123 tempo1"
text4 = " tempo"
print(
  'Matching Text1:', pattern.match(text1),
  '\nMatching Text2:', pattern.match(text2),
  '\nMatching Text3:', pattern.match(text3),
  '\nMatching Text4:', pattern.match(text4))
```

```
#:> Matching Text1: <re.Match object; span=(0, 5), match='tempo'>
```

```
#:> Matching Text2: <re.Match object; span=(0, 5), match='tempo'>
#:> Matching Text3: None
#:> Matching Text4: None
```

17.1.3.2 Special Sequence

```
. : [^\n]
\d: [0-9]              \D: [^0-9]
\s: [ \t\n\r\f\v]      \S: [^ \t\n\r\f\v]
\w: [a-zA-Z0-9_]       \W: [^a-zA-Z0-9_]
\t: tab
\n: newline
\b: word boundry (delimited by space, \t, \n)
```

Word Boundary Using `\b`:

- `\bABC` match if specified characters at the beginning of word (delimited by space, `\t`, `\n`), or beginning of newline

- `ABC\b` match if specified characters at the end of word (delimited by space, `\t`, `\n`), or end of the line

```
text = "ABCD ABC XYZABC"
pattern1 = re.compile(r'\bABC')
pattern2 = re.compile(r'ABC\b')
pattern3 = re.compile(r'\bABC\b')

print('Match word that begins ABC:',
  pattern1.findall(text), '\n',
  'Match word that ends with ABC:',
  pattern2.findall(text),'\n',
  'Match isolated word with ABC:',
  pattern3.findall(text))
```

```
#:> Match word that begins ABC: ['ABC', 'ABC']
#:>  Match word that ends with ABC: ['ABC', 'ABC']
#:>  Match isolated word with ABC: ['ABC']
```

17.1.3.3 Repetition

When repetition is used, re will be **greedy**; it try to repeat as many times as possible. If **later portions of the pattern don't match**, the matching engine will then **back up and try again** with fewer repetitions.

```
?:    zero or 1 occurance
*:    zero or more occurance
+:    one  or more occurance
```

? Zero or 1 Occurance

```
text = 'abcbcdd'
pattern = re.compile(r'a[bcd]?b')
pattern.findall(text)
```

```
#:> ['ab']
```

+ At Least One Occurance

```
text = 'abcbcdd'
pattern = re.compile(r'a[bcd]+b')
pattern.findall(text)
```

```
#:> ['abcb']
```

*** Zero Or More Occurance Occurance**

```
text = 'abcbcdd'
pattern = re.compile(r'a[bcd]*b')
pattern.findall(text)
```

```
#:> ['abcb']
```

17.1.3.4 Greedy vs Non-Greedy

- The `*`, `+`, and `?` qualifiers are all greedy; they match as much text as possible
- If the <.*> is matched against <a> b <c>, it will match the entire string, and not just <a>
- Adding `?` after the qualifier makes it perform the match in non-greedy; as few characters as possible will be matched. Using the RE <.*?> will match only ‘’

```
text = '<a> ali baba <c>'
greedy_pattern     = re.compile(r'<.*>')
non_greedy_pattern = re.compile(r'<.*?>')
print( 'Greedy:      ' ,        greedy_pattern.findall(text), '\n',
       'Non Greedy: ', non_greedy_pattern.findall(text) )
```

```
#:> Greedy:       ['<a> ali baba <c>']
#:>  Non Greedy:  ['<a>', '<c>']
```

17.1.4 Grouping

When `()` is used in the pattern, retrive the grouping components in MatchObject with `.groups()`. Result is in list. Example below extract hours, minutes and am/pm into a list.

17.1.4.1 Capturing Group

```
text = 'Today at Wednesday, 10:50pm, we go for a walk'
pattern = re.compile(r'(\d\d):(\d\d)(am|pm)')
m = pattern.search(text)
print(
  'All Gropus: ', m.groups(), '\n',
  'Group 1: ', m.group(1), '\n',
  'Group 2: ', m.group(2), '\n',
  'Group 3: ', m.group(3) )
```

```
#:> All Gropus:  ('10', '50', 'pm')
#:>  Group 1:  10
#:>  Group 2:  50
#:>  Group 3:  pm
```

17.1.4.2 Non-Capturing Group

Having `(:? )` means don't capture this group

```
text = 'Today at Wednesday, 10:50pm, we go for a walk'
pattern = re.compile(r'(:?\d\d):(?:\d\d)(am|pm)')
m = pattern.search(text)
print(
  'All Gropus: ', m.groups(), '\n',
  'Group 1: ', m.group(1), '\n',
  'Group 2: ', m.group(2) )
```

```
#:> All Gropus:  ('10', 'pm')
#:>  Group 1:  10
#:>  Group 2:  pm
```

17.1.5 Splittitng

Pattern is used to match **delimters**.

17.1.5.1 Use `re.split()`

```
print( re.split('@',  "aa@bb @ cc "), '\n',
       re.split('\|', "aa|bb | cc "), '\n',
       re.split('\n', "sentence1\nsentence2\nsentence3") )
```

```
#:> ['aa', 'bb ', ' cc ']
#:>  ['aa', 'bb ', ' cc ']
#:>  ['sentence1', 'sentence2', 'sentence3']
```

17.1.5.2 Use re.compile().split()

```
pattern = re.compile(r"\|")
pattern.split("aa|bb | cc ")
```

```
#:> ['aa', 'bb ', ' cc ']
```

17.1.6 Substitution re.sub()

17.1.6.1 Found Match

Example below repalce anything within `{{.*}}`

```
re.sub(r'({{.*}})', 'Durian', 'I like to eat {{Food}}.', flags=re.IGNORECASE)
```

```
#:> 'I like to eat Durian.'
```

Replace `AND` with `&`. This does not require `()` grouping

```
re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
```

```
#:> 'Baked Beans & Spam'
```

17.1.6.2 No Match

If not pattern not found, return the original text.

```
re.sub(r'({{.*}})', 'Durian', 'I like to eat <Food>.', flags=re.IGNORECASE)
```

```
#:> 'I like to eat <Food>.'
```

17.1.7 Practical Examples

17.1.7.1 Extracting Float

```
re_float = re.compile(r'\d+(\.\d+)?')
def extract_float(x):
    money = x.replace(',','')
    result = re_float.search(money)
    return float(result.group()) if result else float(0)

print( extract_float('123,456.78'), '\n',
       extract_float('rm 123.78 (30%)'), '\n',
       extract_float('rm 123,456.78 (30%)') )
```

```
#:> 123456.78
#:>  123.78
#:>  123456.78
```

17.2 Word Tokenizer

17.2.1 Custom Tokenizer

17.2.1.1 Split By Regex Pattern

Use **regex** to split words based on **specific punctuation as delimeter**.
The rule is: split input text when any one or more continuous occurances of specified character.

```
import re
pattern = re.compile(r"[-\s.,;!?]+")
pattern.split("hi @ali--baba, you are aweeeeeesome! isn't it. Believe it.:)")
```

```
#:> ['hi', '@ali', 'baba', 'you', 'are', 'aweeeeeesome', "isn't", 'it', 'Believe', 'it
```

17.2.1.2 Pick By Regex Pattern `nltk.tokenize.RegexpTokenizer`

Any sequence of chars fall within the bracket are considered tokens. Any chars not within the bracket are removed.

```
from nltk.tokenize import RegexpTokenizer
my_tokenizer = RegexpTokenizer(r'[a-zA-Z0-9\']+')
my_tokenizer.tokenize("hi @ali--baba, you are aweeeeeesome! isn't it. Believe it.:")
```

```
#:> ['hi', 'ali', 'baba', 'you', 'are', 'aweeeeeesome', "isn't", 'it', 'Believe', 'it']
```

17.2.2 `nltk.tokenize.word_tokenize()`

Words and punctuations are considered as tokens!

```
import nltk
nltk.download('punkt')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package punkt to /home/msfz751/nltk_data...
#:> [nltk_data]   Package punkt is already up-to-date!
```

```
from nltk.tokenize import word_tokenize
print( word_tokenize("hi @ali-baba, you are aweeeeeesome! isn't it. Believe it.:)") )
```

```
#:> ['hi', '@', 'ali-baba', ',', 'you', 'are', 'aweeeeeesome', '!', 'is', "n't", 'it',
```

17.2.3 `nltk.tokenize.casual.casual_tokenize()`

- Support emoji
- Support reduction of repetition chars
- Support removing userid (@someone)
- Good for social media text

- Punctuations are tokens!

```
from nltk.tokenize.casual     import casual_tokenize
print( casual_tokenize("hi @ali-baba, you are aweeeeeesome! isn't it. Believe it. :)") )
```

```
#:> ['hi', '@ali', '-', 'baba', ',', 'you', 'are', 'aweeeeeesome', '!', "isn't", 'it', '.', 'Beli
```

Example below shorten repeating chars, notice aweeeeeesome becomes aweeesome

```
## shorten repeated chars
print( casual_tokenize("hi @ali-baba, you are aweeeeeesome! isn't it. Believe it.:)",
          reduce_len=True))
```

```
#:> ['hi', '@ali', '-', 'baba', ',', 'you', 'are', 'aweeesome', '!', "isn't", 'it', '.', 'Believe
```

Stripping off User ID

```
## shorten repeated chars, stirp usernames
print( casual_tokenize("hi @ali-baba, you are aweeeeeesome! isn't it. Believe it.:)",
          reduce_len=True,
          strip_handles=True))
```

```
#:> ['hi', '-', 'baba', ',', 'you', 'are', 'aweeesome', '!', "isn't", 'it', '.', 'Believe', 'it',
```

17.2.4 `nltk.tokenize.treebank.TreebankWordTokenizer().tokenize()`

Treebank assume input text is **A sentence**, hence any period combined with word is treated as token.

```
from nltk.tokenize.treebank   import TreebankWordTokenizer
TreebankWordTokenizer().tokenize("hi @ali-baba, you are aweeeeeesome! isn't it. Believe it.:)")
```

```
#:> ['hi', '@', 'ali-baba', ',', 'you', 'are', 'aweeeeeesome', '!', 'is', "n't", 'it.', 'Believe'
```

17.2.5 Corpus Token Extractor

A corpus is a collection of documents (list of documents). A document is a text string containing one or many sentences.

```
from nltk.tokenize import word_tokenize
from nlpia.data.loaders import harry_docs as corpus
```

```
## Tokenize each doc to list, then add to a bigger list
doc_tokens=[]
for doc in corpus:
  doc_tokens += [word_tokenize(doc.lower())]

print('Corpus (Contain 3 Documents):\n',corpus,'\n',
      '\nTokenized result for each document:','\n',doc_tokens)
```

```
#:> Corpus (Contain 3 Documents):
```

```
#:>  ['The faster Harry got to the store, the faster and faster Harry would get home.'
#:>
#:> Tokenized result for each document:
#:>  [['the', 'faster', 'harry', 'got', 'to', 'the', 'store', ',', 'the', 'faster', 'ar
```

Unpack list of token lists from above using sum. To get the **vocabulary** (unique tokens), **convert list to set**.

```
## unpack list of list to list
vocab = sum(doc_tokens,[])
print('\nCorpus Vacabulary (Unique Tokens):\n',
       sorted(set(vocab)))
```

```
#:>
#:> Corpus Vacabulary (Unique Tokens):
#:>  [',', '.', 'and', 'as', 'faster', 'get', 'got', 'hairy', 'harry', 'home', 'is', ':
```

17.3 Sentence Tokenizer

This is about detecting sentence boundry and split text into list of sentences

17.3.1 Sample Text

```
text = '''
Hello Mr. Smith, how are you doing today?
The weather is great, and city is awesome.
The sky is pinkish-blue, Dr. Alba would agree.
You shouldn't eat hard things i.e. cardboard, stones and bushes
'''
```

17.3.2 'nltk.tokenize.punkt.PunktSentenceTokenizer‘

- The `PunktSentenceTokenizer` is an sentence boundary detection algorithm. It is an unsupervised trainable model. This means it can be trained on unlabeled data, aka text that is not split into sentences

- PunkSentneceTokenizer is based on work published on this paepr: Unsupervised Multilingual Sentence Boundary Detection

17.3.2.1 Default Behavior

Vanila tokenizer splits sentences on period `.`, which is not desirable

```
from nltk.tokenize.punkt import PunktSentenceTokenizer, PunktTrainer
#nltk.download('punkt')
tokenizer = PunktSentenceTokenizer()
tokenized_text = tokenizer.tokenize(text)
```

```
for x in tokenized_text:
  print(x)
```

```
#:>
#:> Hello Mr.
#:> Smith, how are you doing today?
#:> The weather is great, and city is awesome.
#:> The sky is pinkish-blue, Dr.
#:> Alba would agree.
#:> You shouldn't eat hard things i.e.
#:> cardboard, stones and bushes
```

17.3.2.2 Pretrained Model - English Pickle

NLTK already includes a pre-trained version of the PunktSentenceTokenizer for English, as you can see, it is quite good

```
tokenizer      = nltk.data.load('tokenizers/punkt/english.pickle')
tokenized_text = tokenizer.tokenize(text)
for x in tokenized_text:
  print(x)
```

```
#:>
#:> Hello Mr. Smith, how are you doing today?
#:> The weather is great, and city is awesome.
#:> The sky is pinkish-blue, Dr. Alba would agree.
#:> You shouldn't eat hard things i.e.
#:> cardboard, stones and bushes
```

17.3.2.3 Adding Abbreviations

- The pretrained tokenizer is not perfect, it wrongly detected 'i.e.' as sentence boundary

- Let's **teach** Punkt by adding the abbreviation to its parameter

Adding Single Abbreviation

```
tokenizer      = nltk.data.load('tokenizers/punkt/english.pickle')

## Add apprevaitions to Tokenizer
tokenizer._params.abbrev_types.add('i.e')

tokenized_text = tokenizer.tokenize(text)
for x in tokenized_text:
  print(x)
```

```
#:>
```

```
#:> Hello Mr. Smith, how are you doing today?
#:> The weather is great, and city is awesome.
#:> The sky is pinkish-blue, Dr. Alba would agree.
#:> You shouldn't eat hard things i.e. cardboard, stones and bushes
```

Add List of Abbreviations

If you have more than one abbreviations, use `update()` with the list of abbreviations

```
from nltk.tokenize.punkt import PunktSentenceTokenizer, PunktParameters

## Add Abbreviations to Tokenizer
tokenizer =  nltk.data.load('tokenizers/punkt/english.pickle')
tokenizer._params.abbrev_types.update(['dr', 'vs', 'mr', 'mrs', 'prof', 'inc', 'i.e'])

sentences = tokenizer.tokenize(text)
for x in sentences:
  print(x)
```

```
#:>
#:> Hello Mr. Smith, how are you doing today?
#:> The weather is great, and city is awesome.
#:> The sky is pinkish-blue, Dr. Alba would agree.
#:> You shouldn't eat hard things i.e. cardboard, stones and bushes
```

17.3.3 `nltk.tokenize.sent_tokenize()`

The `sent_tokenize` function uses an instance of **PunktSentenceTokenizer**, which is already been trained and thus very well knows to mark the end and begining of sentence at what characters and punctuation.

```
from nltk.tokenize import sent_tokenize

sentences = sent_tokenize(text)
for x in sentences:
  print(x)
```

```
#:>
#:> Hello Mr. Smith, how are you doing today?
#:> The weather is great, and city is awesome.
#:> The sky is pinkish-blue, Dr. Alba would agree.
#:> You shouldn't eat hard things i.e. cardboard, stones and bushes
```

17.4 N-Gram

To create n-gram, first create 1-gram token

```
from nltk.util import ngrams
import re
sentence = "Thomas Jefferson began building the city, at the age of 25"
pattern = re.compile(r"[-\s.,;!?]+")
tokens = pattern.split(sentence)
print(tokens)
```

```
#:> ['Thomas', 'Jefferson', 'began', 'building', 'the', 'city', 'at', 'the', 'age', 'of', '25']
```

ngrams() is a generator, therefore, use **list()** to convert into full list

```
ngrams(tokens,2)
```

```
#:> <generator object ngrams at 0x7f127bf22850>
```

Convert 1-gram to 2-Gram, wrap into list

```
grammy = list( ngrams(tokens,2) )
print(grammy)
```

```
#:> [('Thomas', 'Jefferson'), ('Jefferson', 'began'), ('began', 'building'), ('building', 'the'),
```

Combine each 2-gram into a string object

```
[ " ".join(x) for x in grammy]
```

```
#:> ['Thomas Jefferson', 'Jefferson began', 'began building', 'building the', 'the city', 'city a
```

17.5 Stopwords

17.5.1 Custom Stop Words

Build the custom stop words dictionary.

```
stop_words = ['a','an','the','on','of','off','this','is','at']
```

Tokenize text and remove stop words

```
sentence = "The house is on fire"
tokens   = word_tokenize(sentence)
tokens_without_stopwords = [ x for x in tokens if x not in stop_words ]

print(' Original Tokens  : ', tokens, '\n',
      'Removed Stopwords: ',tokens_without_stopwords)
```

```
#:>  Original Tokens  :  ['The', 'house', 'is', 'on', 'fire']
#:>  Removed Stopwords:  ['The', 'house', 'fire']
```

17.5.2 NLTK Stop Words

Contain 179 words, in a list form

```
import nltk
nltk.download('stopwords')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package stopwords to
#:> [nltk_data]     /home/msfz751/nltk_data...
#:> [nltk_data]   Package stopwords is already up-to-date!
```

```
import nltk
#nltk.download('stopwords')
nltk_stop_words = nltk.corpus.stopwords.words('english')
print('Total NLTK Stopwords: ', len(nltk_stop_words),'\n',
      nltk_stop_words)
```

```
#:> Total NLTK Stopwords:  179
#:>  ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "yo
```

17.5.3 SKLearn Stop Words

Contain 318 stop words, in frozenset form

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS as sklearn_stop_words
print(' Total Sklearn Stopwords: ', len(sklearn_stop_words),'\n\n',
       sklearn_stop_words)
```

```
#:>  Total Sklearn Stopwords:  318
#:>
#:>  frozenset({'someone', 'once', 'do', 'whom', 'back', 'how', 'at', 'via', 'cant', 'w
```

17.5.4 Combined NLTK and SKLearn Stop Words

```
combined_stop_words = list( set(nltk_stop_words) | set(sklearn_stop_words) )
print('Total combined NLTK and SKLearn Stopwords:', len( combined_stop_words ),'\n'
      'Stopwords shared among NLTK and SKlearn  :', len( list( set(nltk_stop_words) & s
```

```
#:> Total combined NLTK and SKLearn Stopwords: 378
#:> Stopwords shared among NLTK and SKlearn  : 119
```

17.6 Normalizing

Similar things are combined into single normalized form. This will reduced the vocabulary.

17.6.1 Case Folding

If tokens aren't cap normalized, you will end up with large word list. However, some information is often communicated by capitalization of word, such as name of places. If names are important, consider using proper noun.

```
tokens = ['House','Visitor','Center']
[ x.lower() for x in tokens]
```

```
#:> ['house', 'visitor', 'center']
```

17.6.2 Stemming

- Output of a stemmer is **not necessary a proper word**
- Automatically convert words to **lower cap**
- **Porter stemmer** is a lifetime refinement with 300 lines of python code

- Stemming is faster then Lemmatization

```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
tokens = ('house','Housing','hOuses', 'Malicious','goodness')
[stemmer.stem(x) for x in tokens ]
```

```
#:> ['hous', 'hous', 'hous', 'malici', 'good']
```

17.6.3 Lemmatization

NLTK uses connections within **princeton WordNet** graph for word meanings.

```
nltk.download('wordnet')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package wordnet to /home/msfz751/nltk_data...
#:> [nltk_data]   Package wordnet is already up-to-date!
```

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

print( lemmatizer.lemmatize("better", pos ='a'), '\n',
       lemmatizer.lemmatize("better", pos ='n') )
```

```
#:> good
#:>  better
```

```
print( lemmatizer.lemmatize("good", pos ='a'), '\n',
       lemmatizer.lemmatize("good", pos ='n') )
```

```
#:> good
```

```
#:>  good
```

17.6.4 Comparing Stemming and Lemmatization

- Lemmatization is slower than stemming = Lemmatization is better at retaining meanings
- Lemmatization produce valid english word
- Stemming not necessary produce valid english word
- Both reduce vocabulary size, but increase ambiguity
- For search engine application, stemming and lemmatization will improve recall as it associate more documents with the same query words, however with the cost of reducing precision and accuracy.

For search-based chatbot where accuracy is more important, it should first search with unnormalzied words.

17.7 Wordnet

WordNet® is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations.

WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings. However, there are some important distinctions:
- WordNet interlinks not just word forms–strings of letters–but specific senses of words. As a result, words that are found in close proximity to one another in the network are semantically disambiguated
- WordNet labels the semantic relations among words, whereas the groupings of words in a thesaurus does not follow any explicit pattern other than meaning similarity

Wordnet Princeton

Wordnet Online Browser

17.7.1 NLTK and Wordnet

NLTK (version 3.7.6) includes the English WordNet (147,307 words and 117,659 synonym sets)

```
from nltk.corpus import wordnet as wn

s = set( wn.all_synsets() )
w = set(wn.words())
print('Total words in wordnet  : ' ,   len(w),
      '\nTotal synsets in wordnet: ' , len(s) )
```

```
#:> Total words in wordnet  :  147306
#:> Total synsets in wordnet:  117659
```

17.7.2 Synset

17.7.2.1 Notation

A synset is the basic construct of a word in wordnet. It contains the **Word** itself, with its **POS** tag and **Usage**: **`word.pos.nn`**

```
wn.synset('breakdown.n.03')
```

```
#:> Synset('breakdown.n.03')
```

Breaking down the construct:

```
'breakdown' = Word
'n'         = Part of Speech
'03'        = Usage (01 for most common usage and a higher number would indicate lesser common us
```

17.7.2.2 Part of Speech

Wordnet support five POS tags

```
n - NOUN
v - VERB
a - ADJECTIVE
s - ADJECTIVE SATELLITE
r - ADVERB
```

```
print(wn.ADJ, wn.ADJ_SAT, wn.ADV, wn.NOUN, wn.VERB)
```

```
#:> a s r n v
```

17.7.2.3 Synset Similarity

Let's see how similar are the below two nouns

```
w1 = wn.synset('dog.n.01')
w2 = wn.synset('ship.n.01')
print(w1.wup_similarity(w2))
```

```
#:> 0.4
```

```
w1 = wn.synset('ship.n.01')
w2 = wn.synset('boat.n.01')
print(w1.wup_similarity(w2))
```

```
#:> 0.9090909090909091
```

17.7.3 Synsets

- Synsets is a collection of synsets, which are synonyms that share a common meaning

- A synset (member of Synsets) is identified with a 3-part name of the form:
- A synset can contain one or more lemmas, which represent a specific sense of a specific word

- A synset can contain one or more **Hyponyms and Hypernyms**. These are specific and generalized concepts respectively. For example, 'beach house' and 'guest house' are hyponyms of 'house'. They are more specific concepts of 'house'. And 'house' is a hypernym of 'guest house' because it is the general concept

- **Hyponyms and Hypernyms** are also called lexical relations

```
dogs = wn.synsets('dog') # get all synsets for word 'dog'

for d in dogs:  ## iterate through each Synset
  print(d,':\nDefinition:', d.definition(),
          '\nExample:',    d.examples(),
          '\nLemmas:',     d.lemma_names(),
          '\nHyponyms:',   d.hyponyms(),
          '\nHypernyms:',  d.hypernyms(), '\n\n')
```

```
#:> Synset('dog.n.01') :
#:> Definition: a member of the genus Canis (probably descended from the common wolf)
#:> Example: ['the dog barked all night']
#:> Lemmas: ['dog', 'domestic_dog', 'Canis_familiaris']
#:> Hyponyms: [Synset('basenji.n.01'), Synset('corgi.n.01'), Synset('cur.n.01'), Synse
#:> Hypernyms: [Synset('canine.n.02'), Synset('domestic_animal.n.01')]
#:>
#:>
#:> Synset('frump.n.01') :
#:> Definition: a dull unattractive unpleasant girl or woman
#:> Example: ['she got a reputation as a frump', "she's a real dog"]
#:> Lemmas: ['frump', 'dog']
#:> Hyponyms: []
#:> Hypernyms: [Synset('unpleasant_woman.n.01')]
#:>
#:>
#:> Synset('dog.n.03') :
#:> Definition: informal term for a man
#:> Example: ['you lucky dog']
#:> Lemmas: ['dog']
#:> Hyponyms: []
```

```
#:> Hypernyms: [Synset('chap.n.01')]
#:>
#:>
#:> Synset('cad.n.01') :
#:> Definition: someone who is morally reprehensible
#:> Example: ['you dirty dog']
#:> Lemmas: ['cad', 'bounder', 'blackguard', 'dog', 'hound', 'heel']
#:> Hyponyms: [Synset('perisher.n.01')]
#:> Hypernyms: [Synset('villain.n.01')]
#:>
#:>
#:> Synset('frank.n.02') :
#:> Definition: a smooth-textured sausage of minced beef or pork usually smoked; often served on
#:> Example: []
#:> Lemmas: ['frank', 'frankfurter', 'hotdog', 'hot_dog', 'dog', 'wiener', 'wienerwurst', 'weenie
#:> Hyponyms: [Synset('vienna_sausage.n.01')]
#:> Hypernyms: [Synset('sausage.n.01')]
#:>
#:>
#:> Synset('pawl.n.01') :
#:> Definition: a hinged catch that fits into a notch of a ratchet to move a wheel forward or pre
#:> Example: []
#:> Lemmas: ['pawl', 'detent', 'click', 'dog']
#:> Hyponyms: []
#:> Hypernyms: [Synset('catch.n.06')]
#:>
#:>
#:> Synset('andiron.n.01') :
#:> Definition: metal supports for logs in a fireplace
#:> Example: ['the andirons were too hot to touch']
#:> Lemmas: ['andiron', 'firedog', 'dog', 'dog-iron']
#:> Hyponyms: []
#:> Hypernyms: [Synset('support.n.10')]
#:>
#:>
#:> Synset('chase.v.01') :
#:> Definition: go after with the intent to catch
#:> Example: ['The policeman chased the mugger down the alley', 'the dog chased the rabbit']
#:> Lemmas: ['chase', 'chase_after', 'trail', 'tail', 'tag', 'give_chase', 'dog', 'go_after', 'tr
#:> Hyponyms: [Synset('hound.v.01'), Synset('quest.v.02'), Synset('run_down.v.07'), Synset('tree.
#:> Hypernyms: [Synset('pursue.v.02')]
```

17.8 Part Of Speech (POS)

- In corpus linguistics, part-of-speech tagging (POS tagging or PoS tagging or POST), also called **grammatical tagging** or **word-category disambiguation**, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context–i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph

- This is useful for Information Retrieval, Text to Speech, Word Sense Disambiguation

- The primary target of Part-of-Speech(POS) tagging is to identify the grammatical group of a given word. Whether it is a NOUN, PRONOUN, ADJECTIVE, VERB, ADVERBS, etc. based on the context

- A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc

17.8.1 Tag Sets

- Schools commonly teach that there are 9 parts of speech in English: noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, and interjection

- However, there are clearly many more categories and sub-categories

```
nltk.download('universal_tagset')
```

17.8.1.1 Universal Tagset

This tagset contains **12** coarse tags

```
VERB - verbs (all tenses and modes)
NOUN - nouns (common and proper)
PRON - pronouns
ADJ - adjectives
ADV - adverbs
ADP - adpositions (prepositions and postpositions)
CONJ - conjunctions
DET - determiners
NUM - cardinal numbers
PRT - particles or other function words
X - other: foreign words, typos, abbreviations
. - punctuation
```

17.8.1.2 Penn Treebank Tagset

- This is the most popular "tag set" for American English, developed in the Penn Treebank project

- It has **36 POS tags plus 12** others for punctuations and special symbols

PENN POS Tagset

```
nltk.download('tagsets')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package tagsets to /home/msfz751/nltk_data...
#:> [nltk_data]   Package tagsets is already up-to-date!
```

```
nltk.help.upenn_tagset()
```

```
#:> $: dollar
#:>     $ -$ --$ A$ C$ HK$ M$ NZ$ S$ U.S.$ US$
#:> '': closing quotation mark
#:>     ' ''
#:> (: opening parenthesis
#:>     ( [ {
#:> ): closing parenthesis
#:>     ) ] }
#:> ,: comma
#:>     ,
#:> --: dash
#:>     --
#:> .: sentence terminator
#:>     . ! ?
#:> :: colon or ellipsis
#:>     : ; ...
#:> CC: conjunction, coordinating
#:>     & 'n and both but either et for less minus neither nor or plus so
#:>     therefore times v. versus vs. whether yet
#:> CD: numeral, cardinal
#:>     mid-1890 nine-thirty forty-two one-tenth ten million 0.5 one forty-
#:>     seven 1987 twenty '79 zero two 78-degrees eighty-four IX '60s .025
#:>     fifteen 271,124 dozen quintillion DM2,000 ...
#:> DT: determiner
#:>     all an another any both del each either every half la many much nary
#:>     neither no some such that the them these this those
#:> EX: existential there
#:>     there
#:> FW: foreign word
#:>     gemeinschaft hund ich jeux habeas Haementeria Herr K'ang-
```

```
si vous
#:>     lutihaw alai je jour objets salutaris fille quibusdam pas trop Monte
#:>     terram fiche oui corporis ...
#:> IN: preposition or conjunction, subordinating
#:>     astride among uppon whether out inside pro despite on by throughout
#:>     below within for towards near behind atop around if like until below
#:>     next into if beside ...
#:> JJ: adjective or numeral, ordinal
#:>     third ill-mannered pre-war regrettable oiled calamitous first separable
#:>     ectoplasmic battery-powered participatory fourth still-
to-be-named
#:>     multilingual multi-disciplinary ...
#:> JJR: adjective, comparative
#:>     bleaker braver breezier briefer brighter brisker broader bumper busier
#:>     calmer cheaper choosier cleaner clearer closer colder commoner costlier
#:>     cozier creamier crunchier cuter ...
#:> JJS: adjective, superlative
#:>     calmest cheapest choicest classiest cleanest clearest closest commonest
#:>     corniest costliest crassest creepiest crudest cutest darkest deadliest
#:>     dearest deepest densest dinkiest ...
#:> LS: list item marker
#:>     A A. B B. C C. D E F First G H I J K One SP-44001 SP-
44002 SP-44005
#:>     SP-44007 Second Third Three Two * a b c d first five four one six three
#:>     two
#:> MD: modal auxiliary
#:>     can cannot could couldn't dare may might must need ought shall should
#:>     shouldn't will would
#:> NN: noun, common, singular or mass
#:>     common-carrier cabbage knuckle-duster Casino afghan shed thermostat
#:>     investment slide humour falloff slick wind hyena override subhumanity
#:>     machinist ...
#:> NNP: noun, proper, singular
#:>     Motown Venneboerger Czestochwa Ranzer Conchita Trumplane Christos
#:>     Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI Darryl CTCA
#:>     Shannon A.K.C. Meltex Liverpool ...
#:> NNPS: noun, proper, plural
#:>     Americans Americas Amharas Amityvilles Amusements Anarcho-
Syndicalists
#:>     Andalusians Andes Andruses Angels Animals Anthony Antilles Antiques
#:>     Apache Apaches Apocrypha ...
#:> NNS: noun, common, plural
#:>     undergraduates scotches bric-a-brac products bodyguards facets coasts
#:>     divestitures storehouses designs clubs fragrances averages
#:>     subjectivists apprehensions muses factory-jobs ...
#:> PDT: pre-determiner
```

```
#:>     all both half many quite such sure this
#:> POS: genitive marker
#:>     ' 's
#:> PRP: pronoun, personal
#:>     hers herself him himself hisself it itself me myself one oneself ours
#:>     ourselves ownself self she thee theirs them themselves they thou thy us
#:> PRP$: pronoun, possessive
#:>     her his mine my our ours their thy your
#:> RB: adverb
#:>     occasionally unabatingly maddeningly adventurously professedly
#:>     stirringly prominently technologically magisterially predominately
#:>     swiftly fiscally pitilessly ...
#:> RBR: adverb, comparative
#:>     further gloomier grander graver greater grimmer harder harsher
#:>     healthier heavier higher however larger later leaner lengthier less-
#:>     perfectly lesser lonelier longer louder lower more ...
#:> RBS: adverb, superlative
#:>     best biggest bluntest earliest farthest first furthest hardest
#:>     heartiest highest largest least less most nearest second tightest worst
#:> RP: particle
#:>     aboard about across along apart around aside at away back before behind
#:>     by crop down ever fast for forth from go high i.e. in into just later
#:>     low more off on open out over per pie raising start teeth that through
#:>     under unto up up-pp upon whole with you
#:> SYM: symbol
#:>     % & ' '' ''. ) ). * + ,. < = > @ A[fj] U.S U.S.S.R * ** ***
#:> TO: "to" as preposition or infinitive marker
#:>     to
#:> UH: interjection
#:>     Goodbye Goody Gosh Wow Jeepers Jee-sus Hubba Hey Kee-
reist Oops amen
#:>     huh howdy uh dammit whammo shucks heck anyways whodunnit honey golly
#:>     man baby diddle hush sonuvabitch ...
#:> VB: verb, base form
#:>     ask assemble assess assign assume atone attention avoid bake balkanize
#:>     bank begin behold believe bend benefit bevel beware bless boil bomb
#:>     boost brace break bring broil brush build ...
#:> VBD: verb, past tense
#:>     dipped pleaded swiped regummed soaked tidied convened halted registered
#:>     cushioned exacted snubbed strode aimed adopted belied figgered
#:>     speculated wore appreciated contemplated ...
#:> VBG: verb, present participle or gerund
#:>     telegraphing stirring focusing angering judging stalling lactating
#:>     hankerin' alleging veering capping approaching traveling besieging
#:>     encrypting interrupting erasing wincing ...
#:> VBN: verb, past participle
```

```
#:>     multihulled dilapidated aerosolized chaired languished panelized used
#:>     experimented flourished imitated reunifed factored condensed sheared
#:>     unsettled primed dubbed desired ...
#:> VBP: verb, present tense, not 3rd person singular
#:>     predominate wrap resort sue twist spill cure lengthen brush terminate
#:>     appear tend stray glisten obtain comprise detest tease attract
#:>     emphasize mold postpone sever return wag ...
#:> VBZ: verb, present tense, 3rd person singular
#:>     bases reconstructs marks mixes displeases seals carps weaves snatches
#:>     slumps stretches authorizes smolders pictures emerges stockpiles
#:>     seduces fizzes uses bolsters slaps speaks pleads ...
#:> WDT: WH-determiner
#:>     that what whatever which whichever
#:> WP: WH-pronoun
#:>     that what whatever whatsoever which who whom whosoever
#:> WP$: WH-pronoun, possessive
#:>     whose
#:> WRB: Wh-adverb
#:>     how however whence whenever where whereby whereever wherein whereof why
#:> ``: opening quotation mark
#:>     ` ``
```

17.8.1.3 Claws5 Tagset

Claws5 POS Tagset

```
nltk.help.claws5_tagset()
```

```
#:> AJ0: adjective (unmarked)
#:>     good, old
#:> AJC: comparative adjective
#:>     better, older
#:> AJS: superlative adjective
#:>     best, oldest
#:> AT0: article
#:>     THE, A, AN
#:> AV0: adverb (unmarked)
#:>     often, well, longer, furthest
#:> AVP: adverb particle
#:>     up, off, out
#:> AVQ: wh-adverb
#:>     when, how, why
#:> CJC: coordinating conjunction
#:>     and, or
#:> CJS: subordinating conjunction
#:>     although, when
#:> CJT: the conjunction THAT
```

```
#:>     that
#:> CRD: cardinal numeral
#:>     3, fifty-five, 6609 (excl one)
#:> DPS: possessive determiner form
#:>     your, their
#:> DT0: general determiner
#:>     these, some
#:> DTQ: wh-determiner
#:>     whose, which
#:> EX0: existential THERE
#:>     there
#:> ITJ: interjection or other isolate
#:>     oh, yes, mhm
#:> NN0: noun (neutral for number)
#:>     aircraft, data
#:> NN1: singular noun
#:>     pencil, goose
#:> NN2: plural noun
#:>     pencils, geese
#:> NP0: proper noun
#:>     London, Michael, Mars
#:> NULL: the null tag (for items not to be tagged)
#:> ORD: ordinal
#:>     sixth, 77th, last
#:> PNI: indefinite pronoun
#:>     none, everything
#:> PNP: personal pronoun
#:>     you, them, ours
#:> PNQ: wh-pronoun
#:>     who, whoever
#:> PNX: reflexive pronoun
#:>     itself, ourselves
#:> POS: the possessive (or genitive morpheme)
#:>     's or '
#:> PRF: the preposition OF
#:>     of
#:> PRP: preposition (except for OF)
#:>     for, above, to
#:> PUL: punctuation
#:>     left bracket - ( or [ )
#:> PUN: punctuation
#:>     general mark - . ! , : ; - ? ...
#:> PUQ: punctuation
#:>     quotation mark - ` ' "
#:> PUR: punctuation
#:>     right bracket - ) or ]
```

```
#:> TO0: infinitive marker TO
#:>     to
#:> UNC: "unclassified" items which are not words of the English lexicon
#:> VBB: the "base forms" of the verb "BE" (except the infinitive)
#:>     am, are
#:> VBD: past form of the verb "BE"
#:>     was, were
#:> VBG: -ing form of the verb "BE"
#:>     being
#:> VBI: infinitive of the verb "BE"
#:>     be
#:> VBN: past participle of the verb "BE"
#:>     been
#:> VBZ: -s form of the verb "BE"
#:>     is, 's
#:> VDB: base form of the verb "DO" (except the infinitive)
#:>     do
#:> VDD: past form of the verb "DO"
#:>     did
#:> VDG: -ing form of the verb "DO"
#:>     doing
#:> VDI: infinitive of the verb "DO"
#:>     do
#:> VDN: past participle of the verb "DO"
#:>     done
#:> VDZ: -s form of the verb "DO"
#:>     does
#:> VHB: base form of the verb "HAVE" (except the infinitive)
#:>     have
#:> VHD: past tense form of the verb "HAVE"
#:>     had, 'd
#:> VHG: -ing form of the verb "HAVE"
#:>     having
#:> VHI: infinitive of the verb "HAVE"
#:>     have
#:> VHN: past participle of the verb "HAVE"
#:>     had
#:> VHZ: -s form of the verb "HAVE"
#:>     has, 's
#:> VM0: modal auxiliary verb
#:>     can, could, will, 'll
#:> VVB: base form of lexical verb (except the infinitive)
#:>     take, live
#:> VVD: past tense form of lexical verb
#:>     took, lived
#:> VVG: -ing form of lexical verb
```

```
#:>     taking, living
#:> VVI: infinitive of lexical verb
#:>     take, live
#:> VVN: past participle form of lex. verb
#:>     taken, lived
#:> VVZ: -s form of lexical verb
#:>     takes, lives
#:> XX0: the negative NOT or N'T
#:>     not
#:> ZZ0: alphabetical symbol
#:>     A, B, c, d
```

17.8.1.4 Brown Tagset

Brown POS Tagset

```
nltk.help.brown_tagset()
```

```
#:> (: opening parenthesis
#:>     (
#:> ): closing parenthesis
#:>     )
#:> *: negator
#:>     not n't
#:> ,: comma
#:>     ,
#:> --: dash
#:>     --
#:> .: sentence terminator
#:>     . ? ; ! :
#:> :: colon
#:>     :
#:> ABL: determiner/pronoun, pre-qualifier
#:>     quite such rather
#:> ABN: determiner/pronoun, pre-quantifier
#:>     all half many nary
#:> ABX: determiner/pronoun, double conjunction or pre-quantifier
#:>     both
#:> AP: determiner/pronoun, post-determiner
#:>     many other next more last former little several enough most least only
#:>     very few fewer past same Last latter less single plenty 'nough lesser
#:>     certain various manye next-to-last particular final previous present
#:>     nuf
#:> AP$: determiner/pronoun, post-determiner, genitive
#:>     other's
#:> AP+AP: determiner/pronoun, post-determiner, hyphenated pair
#:>     many-much
```

```
#:> AT: article
#:>     the an no a every th' ever' ye
#:> BE: verb 'to be', infinitive or imperative
#:>     be
#:> BED: verb 'to be', past tense, 2nd person singular or all persons plural
#:>     were
#:> BED*: verb 'to be', past tense, 2nd person singular or all persons plural, negated
#:>     weren't
#:> BEDZ: verb 'to be', past tense, 1st and 3rd person singular
#:>     was
#:> BEDZ*: verb 'to be', past tense, 1st and 3rd person singular, negated
#:>     wasn't
#:> BEG: verb 'to be', present participle or gerund
#:>     being
#:> BEM: verb 'to be', present tense, 1st person singular
#:>     am
#:> BEM*: verb 'to be', present tense, 1st person singular, negated
#:>     ain't
#:> BEN: verb 'to be', past participle
#:>     been
#:> BER: verb 'to be', present tense, 2nd person singular or all persons plural
#:>     are art
#:> BER*: verb 'to be', present tense, 2nd person singular or all persons plural, nega
#:>     aren't ain't
#:> BEZ: verb 'to be', present tense, 3rd person singular
#:>     is
#:> BEZ*: verb 'to be', present tense, 3rd person singular, negated
#:>     isn't ain't
#:> CC: conjunction, coordinating
#:>     and or but plus & either neither nor yet 'n' and/or minus an'
#:> CD: numeral, cardinal
#:>     two one 1 four 2 1913 71 74 637 1937 8 five three million 87-
31 29-5
#:>     seven 1,119 fifty-three 7.5 billion hundred 125,000 1,700 60 100 six
#:>     ...
#:> CD$: numeral, cardinal, genitive
#:>     1960's 1961's .404's
#:> CS: conjunction, subordinating
#:>     that as after whether before while like because if since for than altho
#:>     until so unless though providing once lest s'posin' till whereas
#:>     whereupon supposing tho' albeit then so's 'fore
#:> DO: verb 'to do', uninflected present tense, infinitive or imperative
#:>     do dost
#:> DO*: verb 'to do', uninflected present tense or imperative, negated
#:>     don't
#:> DO+PPSS: verb 'to do', past or present tense + pronoun, personal, nominative, not
```

```
#:>     d'you
#:> DOD: verb 'to do', past tense
#:>     did done
#:> DOD*: verb 'to do', past tense, negated
#:>     didn't
#:> DOZ: verb 'to do', present tense, 3rd person singular
#:>     does
#:> DOZ*: verb 'to do', present tense, 3rd person singular, negated
#:>     doesn't don't
#:> DT: determiner/pronoun, singular
#:>     this each another that 'nother
#:> DT$: determiner/pronoun, singular, genitive
#:>     another's
#:> DT+BEZ: determiner/pronoun + verb 'to be', present tense, 3rd person singular
#:>     that's
#:> DT+MD: determiner/pronoun + modal auxillary
#:>     that'll this'll
#:> DTI: determiner/pronoun, singular or plural
#:>     any some
#:> DTS: determiner/pronoun, plural
#:>     these those them
#:> DTS+BEZ: pronoun, plural + verb 'to be', present tense, 3rd person singular
#:>     them's
#:> DTX: determiner, pronoun or double conjunction
#:>     neither either one
#:> EX: existential there
#:>     there
#:> EX+BEZ: existential there + verb 'to be', present tense, 3rd person singular
#:>     there's
#:> EX+HVD: existential there + verb 'to have', past tense
#:>     there'd
#:> EX+HVZ: existential there + verb 'to have', present tense, 3rd person singular
#:>     there's
#:> EX+MD: existential there + modal auxillary
#:>     there'll there'd
#:> FW-*: foreign word: negator
#:>     pas non ne
#:> FW-AT: foreign word: article
#:>     la le el un die der ein keine eine das las les Il
#:> FW-AT+NN: foreign word: article + noun, singular, common
#:>     l'orchestre l'identite l'arcade l'ange l'assistance l'activite
#:>     L'Universite l'independance L'Union L'Unita l'osservatore
#:> FW-AT+NP: foreign word: article + noun, singular, proper
#:>     L'Astree L'Imperiale
#:> FW-BE: foreign word: verb 'to be', infinitive or imperative
#:>     sit
```

```
#:> FW-BER: foreign word: verb 'to be', present tense, 2nd person singular or all perso
#:>     sind sunt etes
#:> FW-BEZ: foreign word: verb 'to be', present tense, 3rd person singular
#:>     ist est
#:> FW-CC: foreign word: conjunction, coordinating
#:>     et ma mais und aber och nec y
#:> FW-CD: foreign word: numeral, cardinal
#:>     une cinq deux sieben unam zwei
#:> FW-CS: foreign word: conjunction, subordinating
#:>     bevor quam ma
#:> FW-DT: foreign word: determiner/pronoun, singular
#:>     hoc
#:> FW-DT+BEZ: foreign word: determiner + verb 'to be', present tense, 3rd person singu
#:>     c'est
#:> FW-DTS: foreign word: determiner/pronoun, plural
#:>     haec
#:> FW-HV: foreign word: verb 'to have', present tense, not 3rd person singular
#:>     habe
#:> FW-IN: foreign word: preposition
#:>     ad de en a par con dans ex von auf super post sine sur sub avec per
#:>     inter sans pour pendant in di
#:> FW-IN+AT: foreign word: preposition + article
#:>     della des du aux zur d'un del dell'
#:> FW-IN+NN: foreign word: preposition + noun, singular, common
#:>     d'etat d'hotel d'argent d'identite d'art
#:> FW-IN+NP: foreign word: preposition + noun, singular, proper
#:>     d'Yquem d'Eiffel
#:> FW-JJ: foreign word: adjective
#:>     avant Espagnol sinfonica Siciliana Philharmonique grand publique haute
#:>     noire bouffe Douce meme humaine bel serieuses royaux anticus presto
#:>     Sovietskaya Bayerische comique schwarzen ...
#:> FW-JJR: foreign word: adjective, comparative
#:>     fortiori
#:> FW-JJT: foreign word: adjective, superlative
#:>     optimo
#:> FW-NN: foreign word: noun, singular, common
#:>     ballet esprit ersatz mano chatte goutte sang Fledermaus oud def kolkhoz
#:>     roi troika canto boite blutwurst carne muzyka bonheur monde piece force
#:>     ...
#:> FW-NN$: foreign word: noun, singular, common, genitive
#:>     corporis intellectus arte's dei aeternitatis senioritatis curiae
#:>     patronne's chambre's
#:> FW-NNS: foreign word: noun, plural, common
#:>     al culpas vopos boites haflis kolkhozes augen tyrannis alpha-
beta-
#:>     gammas metis banditos rata phis negociants crus Einsatzkommandos
```

```
#:>     kamikaze wohaws sabinas zorrillas palazzi engages coureurs corroborees
#:>     yori Ubermenschen ...
#:> FW-NP: foreign word: noun, singular, proper
#:>     Karshilama Dieu Rundfunk Afrique Espanol Afrika Spagna Gott Carthago
#:>     deus
#:> FW-NPS: foreign word: noun, plural, proper
#:>     Svenskarna Atlantes Dieux
#:> FW-NR: foreign word: noun, singular, adverbial
#:>     heute morgen aujourd'hui hoy
#:> FW-OD: foreign word: numeral, ordinal
#:>     18e 17e quintus
#:> FW-PN: foreign word: pronoun, nominal
#:>     hoc
#:> FW-PP$: foreign word: determiner, possessive
#:>     mea mon deras vos
#:> FW-PPL: foreign word: pronoun, singular, reflexive
#:>     se
#:> FW-PPL+VBZ: foreign word: pronoun, singular, reflexive + verb, present tense, 3rd person sing
#:>     s'excuse s'accuse
#:> FW-PPO: pronoun, personal, accusative
#:>     lui me moi mi
#:> FW-PPO+IN: foreign word: pronoun, personal, accusative + preposition
#:>     mecum tecum
#:> FW-PPS: foreign word: pronoun, personal, nominative, 3rd person singular
#:>     il
#:> FW-PPSS: foreign word: pronoun, personal, nominative, not 3rd person singular
#:>     ich vous sie je
#:> FW-PPSS+HV: foreign word: pronoun, personal, nominative, not 3rd person singular + verb 'to h
#:>     j'ai
#:> FW-QL: foreign word: qualifier
#:>     minus
#:> FW-RB: foreign word: adverb
#:>     bas assai deja um wiederum cito velociter vielleicht simpliciter non zu
#:>     domi nuper sic forsan olim oui semper tout despues hors
#:> FW-RB+CC: foreign word: adverb + conjunction, coordinating
#:>     forisque
#:> FW-TO+VB: foreign word: infinitival to + verb, infinitive
#:>     d'entretenir
#:> FW-UH: foreign word: interjection
#:>     sayonara bien adieu arigato bonjour adios bueno tchalo ciao o
#:> FW-VB: foreign word: verb, present tense, not 3rd person singular, imperative or infinitive
#:>     nolo contendere vive fermate faciunt esse vade noli tangere dites duces
#:>     meminisse iuvabit gosaimasu voulez habla ksu'u'peli'afo lacheln miuchi
#:>     say allons strafe portant
#:> FW-VBD: foreign word: verb, past tense
#:>     stabat peccavi audivi
```

```
#:> FW-VBG: foreign word: verb, present participle or gerund
#:>     nolens volens appellant seq. obliterans servanda dicendi delenda
#:> FW-VBN: foreign word: verb, past participle
#:>     vue verstrichen rasa verboten engages
#:> FW-VBZ: foreign word: verb, present tense, 3rd person singular
#:>     gouverne sinkt sigue diapiace
#:> FW-WDT: foreign word: WH-determiner
#:>     quo qua quod que quok
#:> FW-WPO: foreign word: WH-pronoun, accusative
#:>     quibusdam
#:> FW-WPS: foreign word: WH-pronoun, nominative
#:>     qui
#:> HV: verb 'to have', uninflected present tense, infinitive or imperative
#:>     have hast
#:> HV*: verb 'to have', uninflected present tense or imperative, negated
#:>     haven't ain't
#:> HV+TO: verb 'to have', uninflected present tense + infinitival to
#:>     hafta
#:> HVD: verb 'to have', past tense
#:>     had
#:> HVD*: verb 'to have', past tense, negated
#:>     hadn't
#:> HVG: verb 'to have', present participle or gerund
#:>     having
#:> HVN: verb 'to have', past participle
#:>     had
#:> HVZ: verb 'to have', present tense, 3rd person singular
#:>     has hath
#:> HVZ*: verb 'to have', present tense, 3rd person singular, negated
#:>     hasn't ain't
#:> IN: preposition
#:>     of in for by considering to on among at through with under into
#:>     regarding than since despite according per before toward against as
#:>     after during including between without except upon out over ...
#:> IN+IN: preposition, hyphenated pair
#:>     f'ovuh
#:> IN+PPO: preposition + pronoun, personal, accusative
#:>     t'hi-im
#:> JJ: adjective
#:>     ecent over-all possible hard-fought favorable hard meager fit such
#:>     widespread outmoded inadequate ambiguous grand clerical effective
#:>     orderly federal foster general proportionate ...
#:> JJ$: adjective, genitive
#:>     Great's
#:> JJ+JJ: adjective, hyphenated pair
#:>     big-large long-far
```

```
#:> JJR: adjective, comparative
#:>     greater older further earlier later freer franker wider better deeper
#:>     firmer tougher faster higher bigger worse younger lighter nicer slower
#:>     happier frothier Greater newer Elder ...
#:> JJR+CS: adjective + conjunction, coordinating
#:>     lighter'n
#:> JJS: adjective, semantically superlative
#:>     top chief principal northernmost master key head main tops utmost
#:>     innermost foremost uppermost paramount topmost
#:> JJT: adjective, superlative
#:>     best largest coolest calmest latest greatest earliest simplest
#:>     strongest newest fiercest unhappiest worst youngest worthiest fastest
#:>     hottest fittest lowest finest smallest staunchest ...
#:> MD: modal auxillary
#:>     should may might will would must can could shall ought need wilt
#:> MD*: modal auxillary, negated
#:>     cannot couldn't wouldn't can't won't shouldn't shan't mustn't musn't
#:> MD+HV: modal auxillary + verb 'to have', uninflected form
#:>     shouldda musta coulda must've woulda could've
#:> MD+PPSS: modal auxillary + pronoun, personal, nominative, not 3rd person singular
#:>     willya
#:> MD+TO: modal auxillary + infinitival to
#:>     oughta
#:> NN: noun, singular, common
#:>     failure burden court fire appointment awarding compensation Mayor
#:>     interim committee fact effect airport management surveillance jail
#:>     doctor intern extern night weekend duty legislation Tax Office ...
#:> NN$: noun, singular, common, genitive
#:>     season's world's player's night's chapter's golf's football's
#:>     baseball's club's U.'s coach's bride's bridegroom's board's county's
#:>     firm's company's superintendent's mob's Navy's ...
#:> NN+BEZ: noun, singular, common + verb 'to be', present tense, 3rd person singular
#:>     water's camera's sky's kid's Pa's heat's throat's father's money's
#:>     undersecretary's granite's level's wife's fat's Knife's fire's name's
#:>     hell's leg's sun's roulette's cane's guy's kind's baseball's ...
#:> NN+HVD: noun, singular, common + verb 'to have', past tense
#:>     Pa'd
#:> NN+HVZ: noun, singular, common + verb 'to have', present tense, 3rd person singular
#:>     guy's Knife's boat's summer's rain's company's
#:> NN+IN: noun, singular, common + preposition
#:>     buncha
#:> NN+MD: noun, singular, common + modal auxillary
#:>     cowhand'd sun'll
#:> NN+NN: noun, singular, common, hyphenated pair
#:>     stomach-belly
#:> NNS: noun, plural, common
```

```
#:>     irregularities presentments thanks reports voters laws legislators
#:>     years areas adjustments chambers $100 bonds courts sales details raises
#:>     sessions members congressmen votes polls calls ...
#:> NNS$: noun, plural, common, genitive
#:>     taxpayers' children's members' States' women's cutters' motorists'
#:>     steelmakers' hours' Nations' lawyers' prisoners' architects' tourists'
#:>     Employers' secretaries' Rogues' ...
#:> NNS+MD: noun, plural, common + modal auxillary
#:>     duds'd oystchers'll
#:> NP: noun, singular, proper
#:>     Fulton Atlanta September-October Durwood Pye Ivan Allen Jr. Jan.
#:>     Alpharetta Grady William B. Hartsfield Pearl Williams Aug. Berry J. M.
#:>     Cheshire Griffin Opelika Ala. E. Pelham Snodgrass ...
#:> NP$: noun, singular, proper, genitive
#:>     Green's Landis' Smith's Carreon's Allison's Boston's Spahn's Willie's
#:>     Mickey's Milwaukee's Mays' Howsam's Mantle's Shaw's Wagner's Rickey's
#:>     Shea's Palmer's Arnold's Broglio's ...
#:> NP+BEZ: noun, singular, proper + verb 'to be', present tense, 3rd person singular
#:>     W.'s Ike's Mack's Jack's Kate's Katharine's Black's Arthur's Seaton's
#:>     Buckhorn's Breed's Penny's Rob's Kitty's Blackwell's Myra's Wally's
#:>     Lucille's Springfield's Arlene's
#:> NP+HVZ: noun, singular, proper + verb 'to have', present tense, 3rd person singula
#:>     Bill's Guardino's Celie's Skolman's Crosson's Tim's Wally's
#:> NP+MD: noun, singular, proper + modal auxillary
#:>     Gyp'll John'll
#:> NPS: noun, plural, proper
#:>     Chases Aderholds Chapelles Armisteads Lockies Carbones French Marskmen
#:>     Toppers Franciscans Romans Cadillacs Masons Blacks Catholics British
#:>     Dixiecrats Mississippians Congresses ...
#:> NPS$: noun, plural, proper, genitive
#:>     Republicans' Orioles' Birds' Yanks' Redbirds' Bucs' Yankees' Stevenses'
#:>     Geraghtys' Burkes' Wackers' Achaeans' Dresbachs' Russians' Democrats'
#:>     Gershwins' Adventists' Negroes' Catholics' ...
#:> NR: noun, singular, adverbial
#:>     Friday home Wednesday Tuesday Monday Sunday Thursday yesterday tomorrow
#:>     tonight West East Saturday west left east downtown north northeast
#:>     southeast northwest North South right ...
#:> NR$: noun, singular, adverbial, genitive
#:>     Saturday's Monday's yesterday's tonight's tomorrow's Sunday's
#:>     Wednesday's Friday's today's Tuesday's West's Today's South's
#:> NR+MD: noun, singular, adverbial + modal auxillary
#:>     today'll
#:> NRS: noun, plural, adverbial
#:>     Sundays Mondays Saturdays Wednesdays Souths Fridays
#:> OD: numeral, ordinal
#:>     first 13th third nineteenth 2d 61st second sixth eighth ninth twenty-
```

```
#:>     first eleventh 50th eighteenth- Thirty-ninth 72nd 1/20th twentieth
#:>     mid-19th thousandth 350th sixteenth 701st ...
#:> PN: pronoun, nominal
#:>     none something everything one anyone nothing nobody everybody everyone
#:>     anybody anything someone no-one nothin
#:> PN$: pronoun, nominal, genitive
#:>     one's someone's anybody's nobody's everybody's anyone's everyone's
#:> PN+BEZ: pronoun, nominal + verb 'to be', present tense, 3rd person singular
#:>     nothing's everything's somebody's nobody's someone's
#:> PN+HVD: pronoun, nominal + verb 'to have', past tense
#:>     nobody'd
#:> PN+HVZ: pronoun, nominal + verb 'to have', present tense, 3rd person singular
#:>     nobody's somebody's one's
#:> PN+MD: pronoun, nominal + modal auxillary
#:>     someone'll somebody'll anybody'd
#:> PP$: determiner, possessive
#:>     our its his their my your her out thy mine thine
#:> PP$$: pronoun, possessive
#:>     ours mine his hers theirs yours
#:> PPL: pronoun, singular, reflexive
#:>     itself himself myself yourself herself oneself ownself
#:> PPLS: pronoun, plural, reflexive
#:>     themselves ourselves yourselves
#:> PPO: pronoun, personal, accusative
#:>     them it him me us you 'em her thee we'uns
#:> PPS: pronoun, personal, nominative, 3rd person singular
#:>     it he she thee
#:> PPS+BEZ: pronoun, personal, nominative, 3rd person singular + verb 'to be', present tense, 3r
#:>     it's he's she's
#:> PPS+HVD: pronoun, personal, nominative, 3rd person singular + verb 'to have', past tense
#:>     she'd he'd it'd
#:> PPS+HVZ: pronoun, personal, nominative, 3rd person singular + verb 'to have', present tense,
#:>     it's he's she's
#:> PPS+MD: pronoun, personal, nominative, 3rd person singular + modal auxillary
#:>     he'll she'll it'll he'd it'd she'd
#:> PPSS: pronoun, personal, nominative, not 3rd person singular
#:>     they we I you ye thou you'uns
#:> PPSS+BEM: pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tens
#:>     I'm Ahm
#:> PPSS+BER: pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tens
#:>     we're you're they're
#:> PPSS+BEZ: pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tens
#:>     you's
#:> PPSS+BEZ*: pronoun, personal, nominative, not 3rd person singular + verb 'to be', present ter
#:>     'tain't
#:> PPSS+HV: pronoun, personal, nominative, not 3rd person singular + verb 'to have', uninflected
```

```
#:>     I've we've they've you've
#:> PPSS+HVD: pronoun, personal, nominative, not 3rd person singular + verb 'to have',
#:>     I'd you'd we'd they'd
#:> PPSS+MD: pronoun, personal, nominative, not 3rd person singular + modal auxillary
#:>     you'll we'll I'll we'd I'd they'll they'd you'd
#:> PPSS+VB: pronoun, personal, nominative, not 3rd person singular + verb 'to verb', u
#:>     y'know
#:> QL: qualifier, pre
#:>     well less very most so real as highly fundamentally even how much
#:>     remarkably somewhat more completely too thus ill deeply little overly
#:>     halfway almost impossibly far severly such ...
#:> QLP: qualifier, post
#:>     indeed enough still 'nuff
#:> RB: adverb
#:>     only often generally also nevertheless upon together back newly no
#:>     likely meanwhile near then heavily there apparently yet outright fully
#:>     aside consistently specifically formally ever just ...
#:> RB$: adverb, genitive
#:>     else's
#:> RB+BEZ: adverb + verb 'to be', present tense, 3rd person singular
#:>     here's there's
#:> RB+CS: adverb + conjunction, coordinating
#:>     well's soon's
#:> RBR: adverb, comparative
#:>     further earlier better later higher tougher more harder longer sooner
#:>     less faster easier louder farther oftener nearer cheaper slower tighter
#:>     lower worse heavier quicker ...
#:> RBR+CS: adverb, comparative + conjunction, coordinating
#:>     more'n
#:> RBT: adverb, superlative
#:>     most best highest uppermost nearest brightest hardest fastest deepest
#:>     farthest loudest ...
#:> RN: adverb, nominal
#:>     here afar then
#:> RP: adverb, particle
#:>     up out off down over on in about through across after
#:> RP+IN: adverb, particle + preposition
#:>     out'n outta
#:> TO: infinitival to
#:>     to t'
#:> TO+VB: infinitival to + verb, infinitive
#:>     t'jawn t'lah
#:> UH: interjection
#:>     Hurrah bang whee hmpf ah goodbye oops oh-the-pain-of-
it ha crunch say
#:>     oh why see well hello lo alas tarantara rum-tum-tum gosh hell keerist
```

```
#:>     Jesus Keeeerist boy c'mon 'mon goddamn bah hoo-pig damn ...
#:> VB: verb, base: uninflected present, imperative or infinitive
#:>     investigate find act follow inure achieve reduce take remedy re-
set
#:>     distribute realize disable feel receive continue place protect
#:>     eliminate elaborate work permit run enter force ...
#:> VB+AT: verb, base: uninflected present or infinitive + article
#:>     wanna
#:> VB+IN: verb, base: uninflected present, imperative or infinitive + preposition
#:>     lookit
#:> VB+JJ: verb, base: uninflected present, imperative or infinitive + adjective
#:>     die-dead
#:> VB+PPO: verb, uninflected present tense + pronoun, personal, accusative
#:>     let's lemme gimme
#:> VB+RP: verb, imperative + adverbial particle
#:>     g'ahn c'mon
#:> VB+TO: verb, base: uninflected present, imperative or infinitive + infinitival to
#:>     wanta wanna
#:> VB+VB: verb, base: uninflected present, imperative or infinitive; hypenated pair
#:>     say-speak
#:> VBD: verb, past tense
#:>     said produced took recommended commented urged found added praised
#:>     charged listed became announced brought attended wanted voted defeated
#:>     received got stood shot scheduled feared promised made ...
#:> VBG: verb, present participle or gerund
#:>     modernizing improving purchasing Purchasing lacking enabling pricing
#:>     keeping getting picking entering voting warning making strengthening
#:>     setting neighboring attending participating moving ...
#:> VBG+TO: verb, present participle + infinitival to
#:>     gonna
#:> VBN: verb, past participle
#:>     conducted charged won received studied revised operated accepted
#:>     combined experienced recommended effected granted seen protected
#:>     adopted retarded notarized selected composed gotten printed ...
#:> VBN+TO: verb, past participle + infinitival to
#:>     gotta
#:> VBZ: verb, present tense, 3rd person singular
#:>     deserves believes receives takes goes expires says opposes starts
#:>     permits expects thinks faces votes teaches holds calls fears spends
#:>     collects backs eliminates sets flies gives seeks reads ...
#:> WDT: WH-determiner
#:>     which what whatever whichever whichever-the-hell
#:> WDT+BER: WH-determiner + verb 'to be', present tense, 2nd person singular or all persons plur
#:>     what're
#:> WDT+BER+PP: WH-determiner + verb 'to be', present, 2nd person singular or all persons plural
#:>     whaddya
```

```
#:> WDT+BEZ: WH-determiner + verb 'to be', present tense, 3rd person singular
#:>     what's
#:> WDT+DO+PPS: WH-determiner + verb 'to do', uninflected present tense + pronoun, pers
#:>     whaddya
#:> WDT+DOD: WH-determiner + verb 'to do', past tense
#:>     what'd
#:> WDT+HVZ: WH-determiner + verb 'to have', present tense, 3rd person singular
#:>     what's
#:> WP$: WH-pronoun, genitive
#:>     whose whosever
#:> WPO: WH-pronoun, accusative
#:>     whom that who
#:> WPS: WH-pronoun, nominative
#:>     that who whoever whosoever what whatsoever
#:> WPS+BEZ: WH-pronoun, nominative + verb 'to be', present, 3rd person singular
#:>     that's who's
#:> WPS+HVD: WH-pronoun, nominative + verb 'to have', past tense
#:>     who'd
#:> WPS+HVZ: WH-pronoun, nominative + verb 'to have', present tense, 3rd person singula
#:>     who's that's
#:> WPS+MD: WH-pronoun, nominative + modal auxillary
#:>     who'll that'd who'd that'll
#:> WQL: WH-qualifier
#:>     however how
#:> WRB: WH-adverb
#:>     however when where why whereby wherever how whenever whereon wherein
#:>     wherewith wheare wherefore whereof howsabout
#:> WRB+BER: WH-adverb + verb 'to be', present, 2nd person singular or all persons plu
#:>     where're
#:> WRB+BEZ: WH-adverb + verb 'to be', present, 3rd person singular
#:>     how's where's
#:> WRB+DO: WH-adverb + verb 'to do', present, not 3rd person singular
#:>     howda
#:> WRB+DOD: WH-adverb + verb 'to do', past tense
#:>     where'd how'd
#:> WRB+DOD*: WH-adverb + verb 'to do', past tense, negated
#:>     whyn't
#:> WRB+DOZ: WH-adverb + verb 'to do', present tense, 3rd person singular
#:>     how's
#:> WRB+IN: WH-adverb + preposition
#:>     why'n
#:> WRB+MD: WH-adverb + modal auxillary
#:>     where'd
```

17.8.2 Tagging Techniques

There are few types of tagging techniques:

- Lexical-based

- Rule-based (Brill)
- Probalistic/Stochastic-based (Conditional Random Fields-CRFs, Hidden Markov Models-HMM)
- Neural network-based

NLTK supports the below taggers:

```
from nltk.tag.brill      import BrillTagger
from nltk.tag.hunpos     import HunposTagger
from nltk.tag.stanford   import StanfordTagger, StanfordPOSTagger, StanfordNERTagger
from nltk.tag.hmm        import HiddenMarkovModelTagger, HiddenMarkovModelTrainer
from nltk.tag.senna      import SennaTagger, SennaChunkTagger, SennaNERTagger
from nltk.tag.crf        import CRFTagger
from nltk.tag.perceptron import PerceptronTagger
```

17.8.2.1 nltk `PerceptronTagger`

PerceptronTagger produce tags with **Penn Treebank** tagset

```
from nltk.tag import PerceptronTagger

nltk.download('averaged_perceptron_tagger')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package averaged_perceptron_tagger to
#:> [nltk_data]     /home/msfz751/nltk_data...
#:> [nltk_data]   Package averaged_perceptron_tagger is already up-
to-
#:> [nltk_data]       date!
```

```
tagger = PerceptronTagger()
print('Tagger Classes:', tagger.classes,
      '\n\n# Classes:', len(tagger.classes))
```

```
#:> Tagger Classes: {'RBR', 'IN', "''", 'JJS', 'WRB', 'UH', 'CD', ')', 'VBP', 'RP', ':', 'MD', '(
#:>
#:> # Classes: 45
```

17.8.3 Performing Tagging `nltk.pos_tag()`

Tagging works sentence by sentence:

- Document fist must be splitted into sentences
- Each sentence need to be tokenized into words
- Default NTLK uses `PerceptronTagger`

```
#nltk.download('averaged_perceptron_tagger')
#import nltk
#from nltk.tokenize import word_tokenize, sent_tokenize
doc = '''Sukanya, Rajib and Naba are my good friends. Sukanya is getting married next y

sentences = nltk.sent_tokenize(doc)
for sentence in sentences:
  tokens = nltk.word_tokenize(sentence)
  tagged = nltk.pos_tag(tokens)
  print(tagged)
```

```
#:> [('Sukanya', 'NNP'), (',', ','), ('Rajib', 'NNP'), ('and', 'CC'), ('Naba', 'NNP'),
#:> [('Sukanya', 'NNP'), ('is', 'VBZ'), ('getting', 'VBG'), ('married', 'VBN'), ('next
#:> [('Marriage', 'NN'), ('is', 'VBZ'), ('a', 'DT'), ('big', 'JJ'), ('step', 'NN'), ('i
#:> [('It', 'PRP'), ('is', 'VBZ'), ('both', 'DT'), ('exciting', 'VBG'), ('and', 'CC'),
#:> [('But', 'CC'), ('friendship', 'NN'), ('is', 'VBZ'), ('a', 'DT'), ('sacred', 'JJ')
#:> [('It', 'PRP'), ('is', 'VBZ'), ('a', 'DT'), ('special', 'JJ'), ('kind', 'NN'), ('o
#:> [('Many', 'JJ'), ('of', 'IN'), ('you', 'PRP'), ('must', 'MD'), ('have', 'VB'), ('t
```

17.9 Sentiment

17.9.1 NLTK and Senti-Wordnet

- SentiWordNet **extends Wordnet Synsets** with positive and negative sentiment scores
- The extension was achieved via a complex mix of propagation methods and classifiers. It is thus not a gold standard resource like WordNet (which was compiled by humans), but it has proven useful in a wide range of tasks
- It contains similar number of synsets as wordnet

```
from nltk.corpus import sentiwordnet as swn
nltk.download('sentiwordnet')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package sentiwordnet to
#:> [nltk_data]     /home/msfz751/nltk_data...
#:> [nltk_data]   Package sentiwordnet is already up-to-date!
```

```
s = set( swn.all_senti_synsets() )
print('Total synsets in senti-wordnet  : ' ,   len(s))
```

```
#:> Total synsets in senti-wordnet  :  117659
```

17.9.1.1 Senti-Synset

- Senti-Wordnet extends wordnet with three(3) sentiment scores: positive, negative, objective
- All three scores added up to value 1.0

```
breakdown = swn.senti_synset('breakdown.n.03')
print(
  breakdown, '\n'
  'Positive:', breakdown.pos_score(), '\n',
  'Negative:', breakdown.neg_score(), '\n',
  'Objective:',breakdown.obj_score()
)
```

```
#:> <breakdown.n.03: PosScore=0.0 NegScore=0.25>
#:> Positive: 0.0
#:>  Negative: 0.25
#:>  Objective: 0.75
```

17.9.1.2 Senti-Synsets

Get all the synonmys, with and without the POS information

```
print( list(swn.senti_synsets('slow')), '\n\n',  ## without POS tag
       list(swn.senti_synsets('slow', 'a')) )   ## with POS tag
```

```
#:> [SentiSynset('decelerate.v.01'), SentiSynset('slow.v.02'), SentiSynset('slow.v.03'), SentiSyn
#:>
#:>  [SentiSynset('slow.a.01'), SentiSynset('slow.a.02'), SentiSynset('dense.s.04'), SentiSynset(
```

Get the score for first synset

```
first_synset = list(swn.senti_synsets('slow','a'))[0]

print(
  first_synset, '\n',
  'Positive:',  first_synset.pos_score(), '\n',
  'Negative:',  first_synset.neg_score(), '\n',
  'Objective:', first_synset.obj_score()
)
```

```
#:> <slow.a.01: PosScore=0.0 NegScore=0.0>
#:>  Positive: 0.0
```

```
#:>  Negative: 0.0
#:>  Objective: 1.0
```

17.9.1.3 Converting POS-tag into Wordnet POS-tag

Using Function

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import wordnet as wn

def penn_to_wn(tag):
    """
    Convert between the PennTreebank tags to simple Wordnet tags
    """
    if tag.startswith('J'):
        return wn.ADJ
    elif tag.startswith('N'):
        return wn.NOUN
    elif tag.startswith('R'):
        return wn.ADV
    elif tag.startswith('V'):
        return wn.VERB
    return None

wt = word_tokenize("Star Wars is a wonderful movie")
penn_tags = nltk.pos_tag(wt)
wordnet_tags = [ (x, penn_to_wn(y)) for (x,y) in penn_tags ]

print(
'Penn Tags    :', penn_tags,
'\nWordnet Tags :', wordnet_tags)
```

```
#:> Penn Tags    : [('Star', 'NNP'), ('Wars', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('wor
#:> Wordnet Tags : [('Star', 'n'), ('Wars', 'n'), ('is', 'v'), ('a', None), ('wonderful
```

Using defaultdict

```
import nltk
from nltk.corpus import wordnet as wn
from nltk import word_tokenize, pos_tag
from collections import defaultdict

tag_map = defaultdict(lambda : None)
tag_map['J'] = wn.ADJ
tag_map['R'] = wn.ADV
tag_map['V'] = wn.VERB
tag_map['N'] = wn.NOUN
```

```
wt = word_tokenize("Star Wars is a wonderful movie")
penn_tags = nltk.pos_tag(wt)
wordnet_tags = [ (x, tag_map[y[0]]) for (x,y) in penn_tags ]

print(
'Penn Tags    :', penn_tags,
'\nWordnet Tags :', wordnet_tags)
```

```
#:> Penn Tags    : [('Star', 'NNP'), ('Wars', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('wonderful', '
#:> Wordnet Tags : [('Star', 'n'), ('Wars', 'n'), ('is', 'v'), ('a', None), ('wonderful', 'a'), (
```

17.9.2 Vader

- It is a rule based sentiment analyzer, contain 7503 lexicons
- It is good for **social media** because lexicon contain **emoji and short** form text
- Contain only **3 n-gram**
- Supported by NTLK or install vader seperately (pip install vaderSentiment)

17.9.2.1 Vader Lexicon

The lexicon is a dictionary. To make it iterable, need to convert into list:
- Step 1: Convert `dict` to `dict_items`, which is a list containing items, each item is one dict
- Step 2: Unpack `dict_items` to `list`

```
#from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer   ## seperate pip installec
from nltk.sentiment.vader import SentimentIntensityAnalyzer

nltk.download('vader_lexicon')
```

```
#:> True
#:>
#:> [nltk_data] Downloading package vader_lexicon to
#:> [nltk_data]     /home/msfz751/nltk_data...
#:> [nltk_data]   Package vader_lexicon is already up-to-date!
```

```
vader_lex  = SentimentIntensityAnalyzer().lexicon  # get the lexicon dictionary
vader_list = list(vader_lex.items())               # convert to items then list
print( 'Total Vader Lexicon:', len(vader_lex),'\n',
        vader_list[1:10], vader_list[220:240] )
```

```
#:> Total Vader Lexicon: 7502
#:>  [('%)', -0.4), ('%-)', -1.5), ('&-:', -0.4), ('&:', -0.7), ("( '}{' )", 1.6), ('(%', -
0.9), ("('-:", 2.2), ("(':", 2.3), ('((-:', 2.1)] [('b^d', 2.6), ('cwot', -
2.3), ("d-':", -2.5), ('d8', -3.2), ('d:', 1.2), ('d:<', -3.2), ('d;', -
2.9), ('d=', 1.5), ('doa', -2.3), ('dx', -3.0), ('ez', 1.5), ('fav', 2.0), ('fcol', -
```

```
1.8), ('ff', 1.8), ('ffs', -2.8), ('fkm', -2.4), ('foaf', 1.8), ('ftw', 2.0), ('fu', -
3.7), ('fubar', -3.0)]
```

There is only four N-Gram in the lexicon

```
print('List of N-grams: ')
```

```
#:> List of N-grams:
```

```
[ (tok,score) for tok, score in vader_list if " " in tok]
```

```
#:> [("( '}{' )", 1.6), ("can't stand", -2.0), ('fed up', -
1.8), ('screwed up', -1.5)]
```

If stemming or lemmatization is used, stem/lemmatize the vader lexicon too

```
[ (tok,score) for tok, score in vader_list if "lov" in tok]
```

```
#:> [('beloved', 2.3), ('lovable', 3.0), ('love', 3.2), ('loved', 2.9), ('lovelies', 2
2.7), ('unloved', -1.9), ('unlovelier', -1.9), ('unloveliest', -
1.9), ('unloveliness', -2.0), ('unlovely', -2.1), ('unloving', -
2.3)]
```

17.9.2.2 Polarity Scoring

Scoring result is a dictionary of:

- neg
- neu
- pos
- compound **neg, neu, pos adds up to 1.0**

Example below shows polarity for two sentences:

```
corpus = ["Python is a very useful but hell difficult to learn",
        ":) :) :("]
for doc in corpus:
  print(doc, '-->', "\n:", SentimentIntensityAnalyzer().polarity_scores(doc) )
```

```
#:> Python is a very useful but hell difficult to learn -->
#:> : {'neg': 0.554, 'neu': 0.331, 'pos': 0.116, 'compound': -
0.8735}
#:> :) :) :( -->
#:> : {'neg': 0.326, 'neu': 0.0, 'pos': 0.674, 'compound': 0.4767}
```

17.10 Feature Representation

17.10.1 The Data

A corpus is a collection of multiple documents. In the below example, each document is represented by a sentence.

```
corpus = [
   'This is the first document, :)',
   'This document is the second document.',
   'And this is a third one',
   'Is this the first document?',
]
```

17.10.2 Frequency Count

Using purely frequency count as a feature will obviously bias on long document (which contain a lot of words, hence words within the document will have very high frequency).

17.10.2.1 + Tokenizer

Default Tokenizer
By default, vectorizer apply tokenizer to select minimum **2-chars alphanumeric words**. Below **train** the vectorizer using **fit_transform()**.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()          # initialize the vectorizer
X   = vec.fit_transform(corpus)  # FIT the vectorizer, return fitted data
print(pd.DataFrame(X.toarray(), columns=vec.get_feature_names()),'\n\n',
      'Vocabulary: ', vec.vocabulary_)
```

```
#:>    and  document  first  is  one  second  the  third  this
#:> 0    0         1      1   1    0       0    1      0     1
#:> 1    0         2      0   1    0       1    1      0     1
#:> 2    1         0      0   1    1       0    0      1     1
#:> 3    0         1      1   1    0       0    1      0     1
#:>
#:>  Vocabulary:  {'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5, 'and': 0
```

Custom Tokenizer
You can use a custom tokenizer, which is a **function that return list of words**. Example below uses nltk RegexpTokenizer function, which retains one or more alphanumeric characters.

```
my_tokenizer = RegexpTokenizer(r'[a-zA-Z0-9\']+')  ## Custom Tokenizer
vec2 = CountVectorizer(tokenizer=my_tokenizer.tokenize) ## custom tokenizer's function
X2   = vec2.fit_transform(corpus)  # FIT the vectorizer, return fitted data
print(pd.DataFrame(X2.toarray(), columns=vec2.get_feature_names()),'\n\n',
      'Vocabulary: ', vec.vocabulary_)
```

```
#:>    a  and  document  first  is  one  second  the  third  this
#:> 0  0    0         1      1   1    0       0    1      0     1
#:> 1  0    0         2      0   1    0       1    1      0     1
```

```
#:> 2  1    1         0      0   1    1       0    0      1     1
#:> 3  0    0         1      1   1    0       0    1      0     1
#:>
#:>  Vocabulary:  {'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5
```

1 and 2-Word-Gram Tokenizer
Use `ngram_range()` to specify range of grams needed.

```
vec3 = CountVectorizer(ngram_range=(1,2))         # initialize the vectorizer
X3   = vec3.fit_transform(corpus)     # FIT the vectorizer, return fitted data
print(pd.DataFrame(X3.toarray(), columns=vec3.get_feature_names()),'\n\n',
      'Vocabulary: ', vec.vocabulary_)
```

```
#:>    and  and this  document  document is  first  ...  third one  this  this document
#:> 0    0         0         1            0      1  ...          0     1              0
#:> 1    0         0         2            1      0  ...          0     1              1
#:> 2    1         1         0            0      0  ...          1     1              0
#:> 3    0         0         1            0      1  ...          0     1              0
#:>
#:>    this is  this the
#:> 0        1         0
#:> 1        0         0
#:> 2        1         0
#:> 3        0         1
#:>
#:> [4 rows x 22 columns]
#:>
#:>  Vocabulary:  {'this': 8, 'is': 3, 'the': 6, 'first': 2, 'document': 1, 'second': 5
```

Apply Trained Vectorizer Once the vectorizer had been trained, you can apply them on new corpus. **Tokens not in the vectorizer vocubulary are ignored**.

```
new_corpus = ["My Name is Charlie Angel", "I love to watch Star Wars"]
XX = vec.transform(new_corpus)
pd.DataFrame(XX.toarray(), columns=vec.get_feature_names())
```

```
#:>    and  document  first  is  one  second  the  third  this
#:> 0    0         0      0   1    0       0    0      0     0
#:> 1    0         0      0   0    0       0    0      0     0
```

17.10.2.2 + Stop Words

Vectorizer can optionally be use with stop words list. Use `stop_words=english` to apply filtering using sklearn built-in stop word. You can replace `english` with other word **list object**.

```
vec4 = CountVectorizer(stop_words='english') ## sklearn stopwords list
X4 = vec4.fit_transform(corpus)
```

```
pd.DataFrame(X4.toarray(), columns=vec4.get_feature_names())
```

```
#:>    document  second
#:> 0         1       0
#:> 1         2       1
#:> 2         0       0
#:> 3         1       0
```

17.10.3 TFIDF

17.10.3.1 Equation

$tf(t,d) =$ occurances of term t in document t$n =$ number of documents$df(t) =$ number of documents containing ter

17.10.3.2 TfidfTransformer

To generate TFIDF vectors, first run `CountVectorizer` to get frequency vector matrix. Then take the output into this transformer.

```
from sklearn.feature_extraction.text import TfidfTransformer

corpus = [
    "apple apple apple apple apple banana",
    "apple apple",
    "apple apple apple banana",
    "durian durian durian"]

count_vec = CountVectorizer()
X = count_vec.fit_transform(corpus)

transformer1 = TfidfTransformer(smooth_idf=False,norm=None)
transformer2 = TfidfTransformer(smooth_idf=False,norm='l2')
transformer3 = TfidfTransformer(smooth_idf=True,norm='l2')

tfidf1 = transformer1.fit_transform(X)
tfidf2 = transformer2.fit_transform(X)
tfidf3 = transformer3.fit_transform(X)

print(
  'Frequency Count: \n', pd.DataFrame(X.toarray(), columns=count_vec.get_feature_names()),
  '\n\nVocabulary: ', count_vec.vocabulary_,
  '\n\nTFIDF Without Norm:\n',tfidf1.toarray(),
  '\n\nTFIDF with L2 Norm:\n',tfidf2.toarray(),
  '\n\nTFIDF with L2 Norm (smooth):\n',tfidf3.toarray())
```

```
#:> Frequency Count:
```

```
#:>    apple  banana  durian
#:> 0      5       1       0
#:> 1      2       0       0
#:> 2      3       1       0
#:> 3      0       0       3
#:>
#:> Vocabulary:  {'apple': 0, 'banana': 1, 'durian': 2}
#:>
#:> TFIDF Without Norm:
#:>  [[6.43841036 1.69314718 0.        ]
#:>  [2.57536414 0.         0.        ]
#:>  [3.86304622 1.69314718 0.        ]
#:>  [0.         0.         7.15888308]]
#:>
#:> TFIDF with L2 Norm:
#:>  [[0.96711783 0.25432874 0.        ]
#:>  [1.         0.         0.        ]
#:>  [0.91589033 0.40142857 0.        ]
#:>  [0.         0.         1.        ]]
#:>
#:> TFIDF with L2 Norm (smooth):
#:>  [[0.97081492 0.23982991 0.        ]
#:>  [1.         0.         0.        ]
#:>  [0.92468843 0.38072472 0.        ]
#:>  [0.         0.         1.        ]]
```

17.10.3.3 TfidfVectorizer

This vectorizer gives end to end processing from corpus into TFIDF vector matrix, including tokenization, stopwords.

```
from sklearn.feature_extraction.text import TfidfVectorizer
my_tokenizer = RegexpTokenizer(r'[a-zA-Z0-9\']+')  ## Custom Tokenizer

vec1 = TfidfVectorizer(tokenizer=my_tokenizer.tokenize,  stop_words='english') #defaul
vec2 = TfidfVectorizer(tokenizer=my_tokenizer.tokenize, stop_words='english',smooth_id
vec3 = TfidfVectorizer(tokenizer=my_tokenizer.tokenize, stop_words='english', norm=None

X1   = vec1.fit_transform(corpus)  # FIT the vectorizer, return fitted data
X2   = vec2.fit_transform(corpus)  # FIT the vectorizer, return fitted data
X3   = vec3.fit_transform(corpus)  # FIT the vectorizer, return fitted data

print(
  'TFIDF Features (Default with Smooth and L2 Norm):\n',
  pd.DataFrame(X1.toarray().round(3), columns=vec1.get_feature_names()),
  '\n\nTFIDF Features (without Smoothing):\n',
```

```
  pd.DataFrame(X2.toarray().round(3), columns=vec2.get_feature_names()),
  '\n\nTFIDF Features (without L2 Norm):\n',
  pd.DataFrame(X3.toarray().round(3), columns=vec3.get_feature_names())
  )
```

```
#:> TFIDF Features (Default with Smooth and L2 Norm):
#:>     apple  banana  durian
#:> 0  0.971   0.240     0.0
#:> 1  1.000   0.000     0.0
#:> 2  0.925   0.381     0.0
#:> 3  0.000   0.000     1.0
#:>
#:> TFIDF Features (without Smoothing):
#:>     apple  banana  durian
#:> 0  0.967   0.254     0.0
#:> 1  1.000   0.000     0.0
#:> 2  0.916   0.401     0.0
#:> 3  0.000   0.000     1.0
#:>
#:> TFIDF Features (without L2 Norm):
#:>     apple  banana  durian
#:> 0  6.116   1.511   0.000
#:> 1  2.446   0.000   0.000
#:> 2  3.669   1.511   0.000
#:> 3  0.000   0.000   5.749
```

17.11 Appliction

17.11.1 Document Similarity

Document1 and Document 2 are mutiplicate of Document0, therefore their consine similarity is the same.

```
documents = (
    "apple apple banana",
    "apple apple banana apple apple banana",
    "apple apple banana apple apple banana apple apple banana")

from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vec = TfidfVectorizer()
tfidf_matrix = tfidf_vec.fit_transform(documents)

from sklearn.metrics.pairwise import cosine_similarity
print('Cosine Similarity betwen doc0 and doc1:\n',cosine_similarity(tfidf_matrix[0], tfidf_matrix
```

```
#:> Cosine Similarity betwen doc0 and doc1:
```

```
#:>  [[1.]]
```

```
print('Cosine Similarity betwen doc1 and doc2:\n',cosine_similarity(tfidf_matrix[1], t
```

```
#:> Cosine Similarity betwen doc1 and doc2:
#:>  [[1.]]
```

```
print('Cosine Similarity betwen doc1 and doc2:\n',cosine_similarity(tfidf_matrix[0], t
```

```
#:> Cosine Similarity betwen doc1 and doc2:
#:>  [[1.]]
```

17.12 Naive Bayes

17.12.1 Libraries

```
from nlpia.data.loaders import get_data
from nltk.tokenize.casual     import casual_tokenize
from collections import Counter
```

17.12.2 The Data

```
movies = get_data('hutto_movies')   # download data
print(movies.head(), '\n\n',
      movies.describe())
```

```
#:>     sentiment                                               text
#:> id
#:> 1    2.266667  The Rock is destined to be the 21st Century's ...
#:> 2    3.533333  The gorgeously elaborate continuation of ''The...
#:> 3   -0.600000                     Effective but too tepid biopic
#:> 4    1.466667  If you sometimes like to go to the movies to h...
#:> 5    1.733333  Emerges as something rare, an issue movie that...
#:>
#:>            sentiment
#:> count  10605.000000
#:> mean       0.004831
#:> std        1.922050
#:> min       -3.875000
#:> 25%       -1.769231
#:> 50%       -0.080000
#:> 75%        1.833333
#:> max        3.941176
```

17.12.3 Bag of Words

- Tokenize each record, remove single character token, then convert into list of counters (words-frequency pair).
- Each item in the list is a counter, which represent word frequency within the record

```
bag_of_words = []
for text in movies.text:
    tokens = casual_tokenize(text, reduce_len=True, strip_handles=True)  # tokenize
    tokens = [x for x in tokens if len(x)>1]                          ## remove single char token
    bag_of_words.append( Counter(tokens, strip_handles=True)  ## add to our BoW
    )

unique_words =  list( set([ y  for x in bag_of_words  for y in x.keys()]) )

print("Total Rows: ", len(bag_of_words),'\n\n',
      'Row 1 BoW: ',bag_of_words[:1],'\n\n',    # see the first two records
      'Row 2 BoW: ', bag_of_words[:2], '\n\n',
      'Total Unique Words: ', len(unique_words))
```

```
#:> Total Rows:  10605
#:>
#:>  Row 1 BoW:  [Counter({'to': 2, 'The': 1, 'Rock': 1, 'is': 1, 'destined': 1, 'be': 1, 'the':
#:>
#:>  Row 2 BoW:  [Counter({'to': 2, 'The': 1, 'Rock': 1, 'is': 1, 'destined': 1, 'be': 1, 'the':
#:>
#:>  Total Unique Words:  20686
```

Convert NaN into 0 then all features into integer

```
bows_df = pd.DataFrame.from_records(bag_of_words)
bows_df = bows_df.fillna(0).astype(int)  # replace NaN with 0, change to integer
bows_df.head()
```

```
#:>    The  Rock  is  destined  to  ...  Bearable  Staggeringly  ve  muttering  dissing
#:> 0    1     1   1         1   2  ...         0             0   0          0        0
#:> 1    2     0   1         0   0  ...         0             0   0          0        0
#:> 2    0     0   0         0   0  ...         0             0   0          0        0
#:> 3    0     0   1         0   4  ...         0             0   0          0        0
#:> 4    0     0   0         0   0  ...         0             0   0          0        0
#:>
#:> [5 rows x 20686 columns]
```

17.12.4 Build The Model

```
from sklearn.naive_bayes import MultinomialNB
train_y  = movies.sentiment>0   # label
train_X  = bows_df              # features
nb_model = MultinomialNB().fit( train_X, train_y)
```

17.12.5 Train Set Prediction

First, make a prediction on training data, then compare to ground truth.

```
train_predicted = nb_model.predict(bows_df)
print("Accuracy: ", np.mean(train_predicted==train_y).round(4))
```

```
#:> Accuracy:  0.9357
```

Chapter 18

Web Scrapping

18.1 requests

18.1.1 Creating A Session

```
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
import random


_retries = Retry(connect=10,read=10,backoff_factor=1)   # backoff is incremental interval in seco
_timeout = (10,10)  ## connect, read timeout in seconds

rqs = requests.Session()
rqs.mount( 'http://' ,  HTTPAdapter(max_retries= _retries))
rqs.mount( 'https://' , HTTPAdapter(max_retries= _retries))
```

```
link1 = 'https://www.yahoo.com'
link2 = 'http://mamamia777.com.au'
#user_agent = {'User-Agent': random.choice(_USER_AGENTS)}
#response1  = rqs.get(link1, timeout=_timeout)
#response2  = rqs.get(link2, timeout=_timeout)
```

```
print (page1.status_code)
```

18.1.2 Rotating Broswer

```
_USER_AGENTS = [
   #Chrome
```

```
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) C
    'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
    'Mozilla/5.0 (Windows NT 5.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
    'Mozilla/5.0 (Windows NT 6.2; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
    'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0
    'Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) C
    'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) C
    'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Ch
    #Firefox
    'Mozilla/4.0 (compatible; MSIE 9.0; Windows NT 6.1)',
    'Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko',
    'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)',
    'Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko',
    'Mozilla/5.0 (Windows NT 6.2; WOW64; Trident/7.0; rv:11.0) like Gecko',
    'Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko',
    'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0; Trident/5.0)',
    'Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko',
    'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)',
    'Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko',
    'Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/6.0)',
    'Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)',
    'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.5072
```

18.2 BeautifulSoup

18.2.1 Module Import

```
from bs4 import BeautifulSoup
```

18.2.2 HTML Tag Parsing

18.2.2.1 Sample Data

```
my_html = '''
<div id="my-id1" class='title'>
    <p>This Is My Title</p>

    <div id="my-id2" class='subtitle' custom_attr='funny'>
        <p>This is Subtitle</p>
    </div>

    <div id="my-id3" class='title',   custom_attr='funny'>
```

```
        <p>This is paragraph1</p>
        <p>This is paragraph2</p>
        <h3>This is paragraph3</h3>
    </div>
</div>
'''
soup = BeautifulSoup(my_html)
```

18.2.2.2 First Match

ID Selector

Everthing under the selected tag will be returned.

```
soup.find(id='my-id1')
```

```
#:> <div class="title" id="my-id1">
#:> <p>This Is My Title</p>
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>
#:> <div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>
#:> </div>
```

Class Selector

```
soup.find(class_='subtitle')
```

```
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>
```

Attribute Selector

```
soup.find(custom_attr='funny')
```

```
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>
```

```
soup.find(       custom_attr='funny')
```

```
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>
```

```
soup.find('div', custom_attr='funny')
```

```
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>
```

18.2.2.3 Find All Matches

find_all

```
soup = BeautifulSoup(my_html)
multiple_result = soup.find_all(class_='title')
print( 'Item 0: \n',     multiple_result[0],
       '\n\nItem 1: \n', multiple_result[1])
```

```
#:> Item 0:
#:>  <div class="title" id="my-id1">
#:> <p>This Is My Title</p>
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>
#:> <div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>
#:> </div>
#:>
#:> Item 1:
#:>  <div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>
```

CSS Selector using `select()`

Above can be achieved using css selector. It return an array of result (multiple matches).

```
multiple_result = soup.select('.title')
print( 'Item 0: \n',     multiple_result[0],
       '\n\nItem 1: \n', multiple_result[1])
```

```
#:> Item 0:
#:>  <div class="title" id="my-id1">
#:> <p>This Is My Title</p>
#:> <div class="subtitle" custom_attr="funny" id="my-id2">
```

```
#:> <p>This is Subtitle</p>
#:> </div>
#:> <div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>
#:> </div>
#:>
#:> Item 1:
#:>  <div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>
```

More granular exmaple of css selector.

```
soup.select('#my-id1 div.subtitle')
```

```
#:> [<div class="subtitle" custom_attr="funny" id="my-id2">
#:> <p>This is Subtitle</p>
#:> </div>]
```

Using **contains()**

```
soup.select("p:contains('This is paragraph')")
```

```
#:> [<p>This is paragraph1</p>, <p>This is paragraph2</p>]
```

Combining ID, Class and Custom Attribute in the selector

```
soup.select("div#my-id3.title[custom_attr='funny']:contains('This is paragraph')")
```

```
#:> [<div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>]
```

18.2.3 Meta Parsing

```
my_meta = '''
<meta property="description"   content="KUALA LUMPUR: blah blah"   category="Malaysia">
<meta property="publish-date"  content="2012-01-03">
'''
soup = BeautifulSoup(my_meta)
soup.find('meta', property='description')['content']
```

```
#:> 'KUALA LUMPUR: blah blah'
```

```
soup.find('meta', property='description')['category']
```

```
#:> 'Malaysia'
```

```
soup.find('meta', property='publish-date')['content']
```

```
#:> '2012-01-03'
```

```
soup.find('meta', category='Malaysia')['property']
```

```
#:> 'description'
```

18.2.4 Getting Content

18.2.4.1 Get Content `get_text(strip=, separator=)`

- Use **`strip=True`** to strip whitespace from the beginning and end of each bit of text

- Use **'separator='\n'** to specify a string to be used to join the bits of text together
- It is recommended to use `strip=True, separator='\n'` so that result from different operating system will be consistant

```
soup = BeautifulSoup(my_html)
elem = soup.find(id = "my-id3")
elem.get_text(strip=False)
```

```
#:> '\nThis is paragraph1\nThis is paragraph2\nThis is paragraph3\n'
```

- strip=True combine with separator will retain only the user readable text portion of each tag, with separator seperating them

```
elem.get_text(strip=True, separator='\n')
```

```
#:> 'This is paragraph1\nThis is paragraph2\nThis is paragraph3'
```

18.2.4.2 Splitting Content

It is useful to split using separator into list of string.

```
elem = soup.find(id = "my-id3")
elem.get_text(strip=True, separator='\n').split('\n')
```

```
#:> ['This is paragraph1', 'This is paragraph2', 'This is paragraph3']
```

18.2.5 Traversing

18.2.5.1 Get The Element

```
elems = soup.select("div#my-id3.title[custom_attr='funny']:contains('This is paragraph')")
elem = elems[0]
elem
```

```
#:> <div class="title" custom_attr="funny" id="my-id3">
#:> <p>This is paragraph1</p>
#:> <p>This is paragraph2</p>
#:> <h3>This is paragraph3</h3>
#:> </div>
```

18.2.5.2 Traversing Children

All Children In List `findChildren()`

```
elem.findChildren()
```

```
#:> [<p>This is paragraph1</p>, <p>This is paragraph2</p>, <h3>This is paragraph3</h3>]
```

Next Children `findNext()`

- If the element has children, this will get the immediate child
- If the element has no children, this will find the next element in the hierechy

```
first_child = elem.fin
print(
elem.findNext().get_text(strip=True), '\n',
elem.findNext().findNext().get_text(strip=True), '\n')
```

```
#:> This is paragraph1
#:>  This is paragraph2
```

18.2.5.3 Traversing To Parent `parent()`

```
elem_parent = elem.parent
elem_parent.attrs
```

```
#:> {'id': 'my-id1', 'class': ['title']}
```

18.2.5.4 Get The Sibling `findPreviousSibling()`

Sibling is element at the same level of hierachy

```
elem_prev_sib = elem.findPreviousSibling()
elem_prev_sib.attrs
```

```
#:> {'id': 'my-id2', 'class': ['subtitle'], 'custom_attr': 'funny'}
```