# Command line

python -h shows all options

-I run script but do so interactively so you can debug an error

-v verbose

-vv very verbose

python --help-env describes the env variables (PYTHONPATH, PYTHONHOME,etc..) ( not the values )

python -c print('Hello')

python -m run a module as a script

python -c run a program

python -d debug

python -V print version same as --version

set tells me

PYTHONPATH=C:\Users\arono\anaconda3\pkgs\mysql-connector-python-8.0.18-py36h13ed8b8\_1\Lib\site-packages

python -m pip parameters include install, download, list, show, config

py -m pip list -v see everything

!pip install modsimpy you can run this in jupyter

!pip install Quantlib spyder too

os.getcwd()

os.chdir(r"C:\Users\arono\source\python")

del my\_var # remove my\_var from environment

# file IO

f = open("C:\\Users\\arono\\idle\_env.txt", "w")

f.write("\_\_import\_\_")

f.close()

import csv

thefile="C:\\Users\\arono\\Documents\\R\curves.csv"

with open(thefile) as csv\_file:

csv\_reader = csv.reader(thefile, delimiter=',')

for row in csv\_reader:

if ((line\_count > 0 ) and int(row[1] < 379 )):

cterm=int(row[1])

infile = open("C:\\_InProgress\Work\Math\CUNY\PythonMath\Top100\_RunningBacks.txt")

allnumbs = infile.readlines()

num\_list=[]

for line in allnumbs:

num\_list.append(int(line.strip()))

# data types

List, Tuple, Set and Dictionary are the 4 core containers. Only a Tuple is immutable.

List, Set, Dictionary and Data Frame contain references, not native data.

Core native scalars are strings, and numbers (integer,float,complex) and booleans. These are all immutable.

If you update a string in a dataframe you are still creating a new string somewhere, but the dataframe gets updated with a new reference.

Immutable, meaning you can only allocate and reallocate the memory, you cant pass by reference, so bottom line don’t build up big tuples, don’t pass big tuples to functions. Constantly updating floats can lead to problems if GC cant catch up fast enough.

The rest are mutable.

Pandas provides dataframes and series which are like database tables in which every column and every row has a label

A series is like a DataFrame with a single column

Dictionary key-value pairs, a.k.a associative array or hash

Mutable mappings of object references

newdict = {}

colon : defines the key:value

methods include keys, items, values, pop(key)

**Tuple** newtuple = ()

"sequences of object references"

accessed by index like an array of pointers pointing to different data types

immutable

not many methods…has count()

List Mutable

newlist = []

Methods include append, insert, sort

Set Mutable

newset = set()

newset = {"Islanders", "Sabres"}

unordered

may not contain duplicates

methods include add,update,remove

There are alternative such as SortedDictionary and FrozenSet

ndarray numpy

s.ndim, s.shape, s.size, s.dtype to see the dimensions, shape,size, data type

Series pandas, similar to numpy.ndarray

type(b) is Test1 returns True if b is an instance of Class1

isinstance(b, Class1) returns True if b is an instance of Class1 or one of its subclasses

a = 1+ 3j <class 'complex'>

dir(random) # see whats in random

help(random.expovariate)

x.\_\_dir\_\_() list of names in the current local scope OR attributes of a method

x.\_\_str\_\_() displays data in a single string representation

x.\_\_module\_\_ gives the module your object belongs to

locals(),vars(),globals() all return a whole lot besides your variables,

locals returns less if you call it within a restricted namespace

vars() is designed to return \_\_dict\_\_ of the object you pass it

These special attributes exist for objects, instances or modules

object.\_\_dict\_\_ dictionary of all attributes

instance.\_\_class\_\_ the class to which the instance belongs.

module.\_\_file\_\_ the file or folder of this module

module.\_\_doc\_\_ whats inside the triple quotes of the module

module.\_\_name\_\_ the name of the module

one dimensional ndarray array([1950, 1951])

two array([ [1950], [1951]])

triple quotes vs comments ‘’’ can easily enclose multi-line comments

unlike # they also are processed by the interpreter as docstrings (\_\_doc\_\_)

# python 2 vs. 3

3 eliminated some python 2 functions that were redundant

Python2 print statement wasn’t a true function (parenthesis were optional) now in 3 parenthesis are required

# packages

a package is a folder with \_\_init\_\_

a module is any .py file that you import (as long as its in sys.path )

a class is a class, but the \_\_module\_\_ attribute of the class tells you the module it comes from

you can explicitly import a package or module or class or function or variable

# 2 ways to save off all members of a package

saveme=inspect.getmembers(sympy)

saveme=dir(sympy)

with open(r'C:\Users\arono\source\python\Python\getmembers.txt', 'w') as fp:

fp.write("\n".join(str(item) for item in saveme))

import matplotlib.pyplot as plt

means you have to specify plt. before each function, i.e. plt.xlabel('Amount')

from matplotlib.pyplot import barh, yticks, ylabel, xlabel, title, grid, show, plot

now you don’t have to qualify the function, i.e. xlabel('Amount')

import matplotlib.pyplot

now you have to explicitly call, i.e. matplotlib.xlabel('Amount')

dir() # see all variables/functions in your session

dir(modsim) # see all in modsim

print(plt.text.\_\_doc\_\_) called the docstring, its whats inside the 3 ticks, same results as help(plt.text)

modsim.\_\_file\_\_ # see the location of

t.\_\_module\_\_ # numpy

print(inspect.getsource(ModSimSeries)) # see source code, need to import inspect

to install mysql for perl, I ran from the Conda cmd prompt

**conda install -c anaconda mysql-connector-python**

# sys package/environment

a “module” is any file that you can import

a package is a directory with \_\_init\_\_.py in it. When a package is imported, the code is compiled into .pyc files, under a \_\_pycache\_\_ folder

datetime is not a package, it is not a folder, its just datetime.py

datetime.\_\_file\_\_ # C:\Users\arono\anaconda3\lib\datetime.py

pandas is a package

pandas.\_\_file\_\_ # anaconda3\\lib\\site-packages\\pandas\\\_\_init\_\_.py

import goes through sys.path looking for a file or folder by that name

sys.path is current working directory, PYTHONPATH, plus whatever lib\site path is constructed

python -m venv venv calls venv in the venv module, copies python, and modules locally

if (len(sys.argv) != 3):

print("Please pass one integer ")

sys.exit()

k=int(sys.argv[1])

sys.getwindowsversion()

sys.path()

if sys.platform.startswith('linux'):

sys.path\_importer\_cache

sys.modules

sys.path

sys.version

for dist in \_\_import\_\_('pkg\_resources').working\_set:

print(dist.project\_name.replace('Python', ''))

py from windows command line, brings you to a python interpreter

here you can type help on all kinds of stuff

py -m pip show matplotlib

py -m pip list -v

pythonw.exe is used on .pyw files (windowless python) they are run as background jobs

kwargs key word args

args is an array of void pointers

kwargs is an array of void pointers to key, value pairs

def myFun(\*\*kwargs):

for key, value in kwargs.items():

print ("%s == %s" %(key, value))

C:\Users\arono\anaconda3\Scripts\pip3.exe list -v

matches the results of

for dist in \_\_import\_\_('pkg\_resources').working\_set:

print(dist.project\_name.replace('Python', ''))

displays 244 packages

for help on any of these....

**python -m pip show Babel**

py -c "import sys; print(sys.path)"

.pyd files are dll’s

if \_\_name\_\_ = “\_\_main\_\_”

\_\_name\_\_ is set to \_\_main\_\_ for the file that is called directly, otherwise its set to the file name

so this allows you to use your file as both an import and a main program

# dataframes

Series and DataFrames are mostly the same but they have differences.

they both have dtypes, series does not have info() which makes it hard to see the index

fi\_series.dtypes # just gives you the type of the value fi\_df.info() # ok

df.dtypes # see the data types

df.describe() # summary stats, IQR, mean

list(df.columns)

pd.DataFrame.info(df)

df.columns=[‘census’,’un’,’clark’] # rename the columns

df\_5YR.info()

len(df\_5YR)

df\_5YR.shape

df\_5YR.size

**quick way to create :**

orb\_stat\_df = pd.DataFrame([[orb\_mean, orb\_stddev, earth\_z\_score]], columns = ['Mean', 'Std Dev','Earth Z Score'])

with data…

ydf = pd.DataFrame([1965,1966,1967,1968,1969,1970,1971,1972,1973,1974,1975,1976,1977],columns =['Year'])

self.data\_frame=pd.DataFrame() # no columns no rows

df = pd.DataFrame(columns = ['Date', 'Rate5YR', 'Rate10YR', 'Rate30YR']) # columns but no rows

df = df.append({'Date' : d\_date, 'avg' : d\_avg},ignore\_index = True) # add a row

df.loc[len(df)] = [16, 'yahoo']

4 ways to iterate:

for ind in df.index:

print(df['Name'][ind], df['Stream'][ind])

for i in range(len(df)) :

print(df.loc[i, "Name"], df.loc[i, "Age"])

for i in range(len(df)) :

print(df.iloc[i, 0], df.iloc[i, 2])

for index, row in df.iterrows():

print (row["Name"], row["Age"])

loc selects rows and columns with specific labels. iloc selects rows and columns at specific integer positions

Retrieving ONE value

df[5][2]

df[filter]

df.loc['row\_index\_key', 'column\_index\_key']

df.iloc[integer,integer]

df.at['row\_index\_key', 'column\_index\_key']

df.query()

df.index.get\_level\_values('asset\_class')

difference between at() and loc() is at() is designed to find a single value, whereas loc() can handle broad queries...

df.loc[0:4, ['points', 'assists']] # return 2 columns from the first 4 rows

df.loc[2:,['a\_str']] # returns a data frame

df.loc[2,'a\_str'] # returns native value

df.at[1,'a\_str'] # returns native value

df.at[1,1]='yinyang' # this creates a new column

df.at[1,'a\_str']='yingyang' # updates the a\_str column

df.iloc[2,1] # yahoo

Strangely if you index a dataframe directly, its column first, then row.

df['a\_str'][1] # yinyang

if you need to use variables in the index

r=1

c='a\_str'

df[c][r]

for c in df.columns:

for r in df.index.values:

print(df[c][r], end= ' ')

Retrieving MANY values

Here is an example of using a filter (mask) and querying a dataframe with a datetime index.

from datetime import datetime

start = datetime.strptime("1990-01-01","%Y-%m-%d")

end = datetime.strptime("2000-01-01","%Y-%m-%d")

mask = (oil\_cpi\_df['date'] > start) & (oil\_cpi\_df['date'] <= end)

the\_nineties = oil\_cpi\_df.loc[mask]

If you want to plot based on the index...

ax.plot\_date(seven\_years\_df.index,seven\_years\_df['KF\_mean'].values , '-' )

To isolate records based on the index

mask = (seven\_years\_df.index > '2007-09-01')

one\_month\_df=seven\_years\_df.loc[mask]

Here is an example of selecting from a multi index

# select all "2022-08-31"

mval\_ac\_df[mval\_ac\_df.index.get\_level\_values('date') == '2022-08-31']

# select all "Real Estate"

mval\_ac\_df[mval\_ac\_df.index.get\_level\_values('asset\_class') == 'Real Estate']

# select only "2022-08-31" and "Real Estate"

mval\_ac\_df[(mval\_ac\_df.index.get\_level\_values('date') == '2022-08-31') & (mval\_ac\_df.index.get\_level\_values('asset\_class') == 'Real Estate')]

MANIPULATING THE INDEX

resetting an index automatically makes it a columns to replace the index with a column just do

portfolio\_df = portfolio\_df.set\_index(portfolio\_df.Ticker)

to drop an index (resetting it to 0,1,2 etc..) and NOT move it to a column

portfolio\_df.reset\_index(drop=True, inplace=True)

df = df.set\_index(pd.Index([43,22,8]))

df.set\_index(pd.Index([4,7,2]), inplace=True) # To set the index without the "df = ", use in\_place

df.index.name='MyIndex'

df.set\_index(pd.Index(['HoHo', 'HaHa','HeeHee']), inplace=True, append=True) # To not replace the existing index, use append

set the index to a range of dates...

ydf = pd.DataFrame([1965,1966,1967,1968,1969,1970,1971,1972,1973,1974,1975,1976],columns =['Year'])

mask = (ydf['Year'] > 1967) & (ydf['Year'] < 1974)

ydf = ydf.loc[mask]

dates=pd.date\_range(datetime(1968,1,1),datetime(1973,1,1), freq='YS').tolist()

ydf.index=dates

use df.info() or df.dtypes for help

pd.MultiIndex.from\_frame(df) # multiple indexes are a pain

df.index.droplevel() # use drop level to get rid of an row index

df.columns = df.columns.droplevel(1) # drop the column index

ratings\_df=ratings\_df.dropna(subset=['rating']) # remove null ratings

movies\_df.columns = ['id', 'name', 'imdb\_rating', 'lead\_actor'] # rename columns

df = df.rename(columns={'Effective Date': 'Date', 'SOFR Index': 'SOFRIndex'}) # rename 2 columns

people\_df['name'] = people\_df['name'].str.strip() # remove the carriage returns

exo\_days\_df=exo\_days\_df[(exo\_days\_df["planet"]!='COCONUTS-2 b')] # remove 1 record

merged\_df = pd.merge(people\_df,merged\_df, left\_on=['id'],right\_on=['people\_id']) # join 2 df’s

merged\_df = merged\_df[["name\_x", "name\_y","imdb\_rating", "rating"]] # isolate 4 columns

agg\_func\_count = { # summarize

'rating': ['count', 'mean'],

'imdb\_rating':['first']

}

answer\_df = merged\_df.groupby(['Movie']).agg(agg\_func\_count)

answer\_df.columns = ['n\_ratings', 'avg rating', 'imdb rating']

# lambda

lambda, comprehension and generators are 3 alternatives to functions, use them if you find them syntactically convenient or to avoid passing around a large dataset

x = lambda a : a + 10 # x is of type <class 'function'>

print(x(5))

heres 2 examples of list comprehension and lambda

is\_even\_list = [lambda arg=x: arg \* 10 for x in range(1, 5)]

filter\_li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final\_list = list(filter(lambda x: (x % 2 != 0), filter\_li))

# comprehension

comprehension is any time you create a list,set,dict, generator on the fly

newlist = [x for x in fruits if "a" in x]

new\_dict **=** {var:var **\*\*** 3 **for** var **in** input\_list **if** var **%** 2 !**=** 0}

type([x\*\*2 for x in range(5)]) # list

type((x\*\*2 for x in range(5))) # generator (not a tuple, strange behavior…)

type({x\*\*2 for x in range(5)}) # set

type({x:x\*\*2 for x in range(5)}) # dict

# generators

any function with a yield statement returns a generator (better than returning a large dataset)

def firstn(n):

num = 0

while num < n:

yield num # don’t build a list, return each value back to the caller

num += 1

return 4 # never reached

sum(firstn(10)) #45 # sums up each number from 1 to 10 without building up a list

firstn() returns a generator, the return 4 is never called

you can loop through a generator object

for value in firstn(5):

print(value)

# context manager

***with*** provides a scope for an object, the manager is an object with ***\_\_enter\_\_()*** and ***\_\_exit\_\_()*** functions

with expression as target\_var:

do\_something(target\_var)

with io.open(inpathnfile, encoding='utf-8') as infile: # open returns a TextIOBase object

for line in infile: # when this block is over, the *manager* closes the TextIOBase object

the TextIOBase object class defines \_\_enter\_\_() and \_\_exit\_\_(). You can build your own.

the main argument for this is the manager closes the resource for you, even if there is an exception

the os module is another example, you can traverse over directories, and the os object will self close

python 3 supports multiple objects….

with open("input.txt") as in\_file, open("output.txt", "w") as out\_file:

# decorator functions

a function that extends a base function

def base\_func:

…

@base\_func

def base\_func\_and\_more:

note a python class can inherit, i.e.

class Student(Person): # Student class inherits Person class

# regexes

import re

txt = "The rain in Spain"

x = re.search("^The.\*Spain$", txt) true

#Return a list containing every occurrence of "ai":

txt = "The rain in Spain"

x = re.findall("ai", txt) [ ‘ai’,’ai’]

#Check if "Portugal" is in the string:

x = re.findall("Portugal", txt) []

# return the position of the search

x = re.search("\s", txt) 3

print("The first white-space character is located in position:", x.start())

x = re.search("Portugal", txt) None

x = re.split("\s", txt) ['The', 'rain', 'in', 'Spain']

# substitute ( replace ) space with 9

x = re.sub("\s", "9", txt) The9rain9in9Spain

# others

from random import randint

NumPy array and statistic functions includes arange(), random, matrix, linspace, zeros

pickle pickle is a package that serializes/unserializes to/from a byte stream, a.k.a. ***pickling***

export PATH=/cs/vision/anaconda5/bin

export PYTHON\_PATH=$SYBASE/SYABASE\_OCS/python/python34\_64r/lib

export ANACONDA=/cs/vision/anaconda5/bin

import sybpydb

conn.sybpydb.connect(user=”visadmin”, password=”visionRepo@1234”, servername=”VIS\_NY\_UAT”)

cursor = conn.cursor()

cursor.execute(“select cpty, subsid from CptyCode where code=’abc’ “)

rows=cursor.fetchall()

for row in rows:

print ( row.cpty, row.subsid)

curr.close()

conn.close()

import pyodbc

conn\_str=”DRIVER=/cs/vision……/libsybdrvodb.so”, PORT=123, DATABASE=fss, UID=visadmin, PWD=

conn = mysql.connector.connect(host='localhost', database='DATA607', user='root', password=mysql\_password)

if conn.is\_connected():

print('Connected to MySQL database')

cursor = conn.cursor()

cursor.execute("select \* from movies")

movies\_df = pd.DataFrame(cursor.fetchall())

Python containers

# List ordered and changeable. Allows duplicates square brackets []

# Tuple ordered and unchangeable. Allows duplicates parenthesis ()

# Set unordered and unindexed. No duplicates curly brackets {}

# Dictionary unordered, changeable and indexed. No duplicates curly brackets {} with key/value

# Data Frame is part of the Pandas package

tuples are immutable objects and considered more efficient, lists are mutable.

Python argv is k=int(sys.argv[1])

# extra

findstr /s /I "html" \*ipynb to search inside files

where /r . \*ipynb to search file names

type \*xml > all.txt to cat all text files to one

to extract python code only from jupyter notebooks

jupyter nbconvert --to script \*.ipynb

type \*.py > all.txt

findstr /V "^#" all.txt > rall.txt

datasets:

from pydataset import data

data() # show available datasets

titanic = data('titanic')

to install mysql for perl, I ran from the Conda cmd prompt

**conda install -c anaconda mysql-connector-python**