1. Import data from a CSV file to a Table/DataFrame

data = pd.read_csv('파일이름', header=0)

data = pd.read_csv('파일이름', header=None)

data = pd.read_csv('파일이름', name = range(7, 0, -1))

data = pd.read_csv('CC3.SI.csv', header=0, names=range(7,0,-1))

➔ 마지막처럼 쓰면 실제 header는 날아가게 됨.

data=pd.read_csv('CC3.SI.csv', index_col=0)

➔ 이거는 첫번째 칼럼을 row name으로 쓰겠다는 뜻

data=pd.read_csv('CC3.SI.csv', index_col=0, parse_dates=True)

➔ parse_dates가 True이면 자동으로 index를 date 형식으로 바꾸겠다는 뜻임. 리스트가 지정될 수도 있음.

2. Manipulate data in a Table/DataFrame

- Remove/Insert data
- Compare data
- Select data
- Apply (other) operations/functions on data

data.index -> row name

data.columns -> column name

➔ 이 둘은 list-like, array-like objects

data[‘Open’] 이런식으로 불러오면 그 칼럼을 불러오겠다는 뜻. Data type은 series.

data.values

data.drop(data.index[data['Volume']==0],inplace=True)

➔ Volume column의 값이 0인 라인은 전부 날려버리겠다. 여기서 axis=0이면 drop row임. axis=1이면 drop columns. 디폴트는 axis=0(axis = 0은 across the row, axis = 1은 across the column 임)

data['15d']= np.round(data['AdjClose'].rolling(15).mean(), 3)

x[x>0] = 1

➔ 이런 형태로 x라는 series에 값을 assign하는게 가능함.

➔ x[x>0]=range(1, 144) 이런식으로 같은 길이면 range를 assign할 수도 있음.

x.diff

➔ 이전 값에서 다음 값을 계속해서 빼주는 명령어. 아래 위 row 사이의 차를 구한다. 첫 값은 NaN이다. 계산할게 없으므로. 결국 계산된 Series 길이와 레이블은 같다.

y[(y>0) | (y<0)]

➔ Y series에서 0보다 작거나 0보다 큰 값만 불러오는 것. |는 Bit-wise or임.

➔ Element-wise or은 np.logical_or([array1], [array2])

idxSell=y.index[y<0]

➔ y<0을 만족시키는 행의 인덱스를 idxSell에 assign

data.loc[idxSell,'crossSell']=data['Adj Close'][idxSell]

➔ loc은 label indexing. data['Adj Close'][idxSell]는 ['Adj Close'] 행의 [idxSell]열을 불러옴.

3. Visualize data

fig, ax= plt.subplots()

혹은

fig = plt.figure()

ax=plt.axes()

더 복잡한 plot의 경우에는 아래와 같이 가능.

fig1=plt.figure()

ax1=fig1.add_axes([0.1, 0.1, 0.8, 0.8])

ax1=fig1.add_subplot(2, 2, 3)

ax1.plot

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style

```
'b'    # blue markers with default shape
'ro'   # red circles
'g-'   # green solid line
'--'   # dashed line with default color
'k^:'  # black triangle_up markers connected by a dotted line
```

plt.fill_between(x,y1,alpha=0.3, color='k')

➔ x와 y1은 각각 list. 순서대로 점을 이어가면서 그 사이를 모두 채운다.

plt.fill_between(x,y1,y2,alpha=0.3, color='k')

➔ y1과 y2 사이를 칠해준다.

plt.annotate('(%s,%s)' % a, xy=a)

➔ 좌표 찍어주는 명령어. xy는 좌표를 말함. float으로 구성된 tuple 값이 들어와야 한다.

zip

➔ create a sequence of tuples. array값을 합쳐준다.

Numpy and Pandas – ndarrays and DataFrames/Serieses

1. Creation

np.arange(시작, 끝, step)

np.linspace(시작, 끝, 사이 개수)

np.linspace(시작, 끝, 사이 개수)

np.ndarray.reshape(array, int or tuple)

np.ones(tuple) -> np.ones(int)는 에러난다.

np.zeros도 np.ones와 마찬가지임.

둘 간의 차이 알기

✓ x = pd.DataFrame([1, 2, 3])

✓ y = pd.DataFrame([[1, 2, 3]])

➔ 여기서 x+y하면 무슨 결과가 나오나?

➔ Broadcasting 되게 하려면 x.values + y.values 필요

f=[[0.0 for j in range(i+1)] for i in range(N+1)] 가 어떻게 작동하는지 이해하기.

2. Indexing and Slicing

Numpy

x = np.array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

➔ x[:1][:1]을 불러오면 [[0, 1, 2]]가 된다. 첫번째 슬라이싱은 row고

두번째 슬라이싱은 column이고 그런게 아님. 처음 [:1]까지만 하면 [[0, 1, 2]]가 끌려나오고 그 다음 [:1]하면 또 그대로 끌려나옴

➔ 그리고 x[0]과 x[:1]의 차이는 x[0]은 [0, 1, 2]를 가져다주지만 x[:1]는 [[0, 1, 2]]를 가져다준다. Square bracket이 하나 더 생김.

➔ X[:1, :1]하면 array[[0]]이 나온다. 첫번째 행을 먼저 가져오고 그 안에서 또 슬라이싱.

➔ print(x[0::2,1:]), print(x>4), print(x[x>5])

➔ print(x[[1,2,0,0]][0])을 하면 첫번째 인덱싱 때문에 2행, 3행, 1행, 1행이 끌려나오는데 그 중에 [0]을 지정해줬으므로 2행만 프린팅 됨.

print(x[0][[1,2,0,0]]) 순서를 반대로 하면 첫번째 행에서 2번째, 3번째, 1번째, 1번째 숫자를 각각 프린트함

➔ print(x[1::-1,[1,2,0,0]]) , print(x[0::2,1:]) multi-dimensional slicing

➔ 리스트는 f[i,j] 식의 인덱싱을 지원하지 않음.

Pandas

✓ Indexing, slicing 정리

➔ Indexing: column + Series <-> array에서는 row임

➔ Fancy indexing: column + DataFrame <-> array에서는 바깥에 [ ] 더붙은 row임

➔ Slicing : row + DataFrame <-> array에서는 바깥에 [ ] 더붙은 row임

➔ DataFrame에서는 df[1, 1] 식으로는 인덱싱 안됨!

iloc

x=pd.DataFrame(np.arange(16).reshape(4,4),index=[2,1,4,3],columns=list('bcad')) 일 때 아래 결과물 상상해보기. 형태와 데이터 타입

- ✓ print(x)
- ✓ print(x.iloc[2], type(x.iloc[2]))
- ✓ print(x.iloc[[2]], type(x.iloc[[2]]))
- ✓ print(x.iloc[:2], type(x.iloc[:2]))
- ✓ print(x.iloc[[True,False,False,True]], type(x.iloc[[True,False,False,True]]))
- ✓ print(x.iloc[1,1])

loc

위와 같은 dataframe에서 아래 결과물 상상해보기

- ✓ print(x)
- ✓ print(x.loc[2], type(x.loc[2]))
- ✓ print(x.loc[[2]], type(x.loc[[2]]))
- ✓ print(x.loc[:2], type(x.loc[:2])) #여기는 레이블 슬라이싱이므로 2도 포함되어야 함.
- ✓ print(x.loc[[True,False,False,True]],
- ✓ type(x.loc[[True,False,False,True]]))
- ✓ print(x.loc[1,'a'])

3. Assignment

Array vs List

- ✓ 리스트에서 슬라이싱 assignment에서는 iterable을 줘야 함. 갯수는 안맞아도 상관없어. 슬라이싱으로 3개 불러와서 값은 하나만 부여해 줘도 괜찮음.
- ✓ 넘파이 assignment에는 두가지가 있음. 하나는 scalar를 주는 것. 그러면 scalar가 범위 전체에 들어감. 그게 아니면 똑같은 길이의 literable을 주는 것
  - ➔ 리스트에서는 길이가 안맞아도 상관없었지만 array에서는 상관있고, 리스트에서는 scalar값을 못줬지만 array에서는 가능하다.

y = x[:2] 이렇게 x의 일부를 가져온 경우

- ✓ 리스트는 shallow copy가 된다. y의 값을 바꿔도 x에는 영향을 미치지 않는다.
- ✓ Array는 y를 바꾸면 x가 똑같이 바뀐다. 왜냐면 x[~~] = y라고 했을 때 y는 copy되는게 아니라 그냥 x를 가르키는 것일 뿐이기 때문에. 리스트와 컨셉이 다르다.
- ✓ 하지만 팬시인덱싱의 경우는 다르다. Slicing of ndarray returns a view; fancy indexing returns a copy.
- ✓ square bracket을 쓰면 view냐 copy냐가 불명확한 문제가 있음. 이 경우에는 multiple indexing을 하는 것이 좋음.

4. **Some Operations**
5. **Some Functions**

all(iterable)

- ✓ all([1, 2, 3]) : True
- ✓ all([1, 0, 3]) : False
- ✓ All([ ]) : True
- ✓ All([[ ]]) : False

➔ 리스트가 모두 True값을 가지면 True를 리턴한다기보다는 False가 없으면 True를 리턴한다고 보는게 맞는 것 같음.

any(iterable)

- ✓ any([1, 2, 3]) : True
- ✓ any([1, 0, 3]) : True
- ✓ any([ ]) : False
- ✓ any([[ ]]) : False

➔ all, any는 1D array에서만 작동한다. 2D 이상에서는 np.any, np.all을 써야함.

```
print(np.any([[True,False],[True,False]]))

print(np.any([[True,False],[True,False]],axis=0))

print(np.any([[True,False],[True,False]],axis=1))

print(np.any([[True,False],[True,False]],axis=-1))
```

➔ 어떤 결과값이 나오는지 머리속으로 그리기

np.maximum(x, y) -> element-wise로 최대값을 돌려줌. x, y에는 스칼라나 array가 들어옴. Broadcasting 가능.

x in range(5) 는 에러메시지를 띄운다. Np array는 이런 형태의 membership test를 지원하지 않음.

np.isin(x, range(5))

➔ numpy array x에 대해서 element-wise membership test수행

numpy.exp, numpy.sqrt, numpy.mean, numpy.std, numpy.sum, numpy.cumsum,

np.random.randn(3,4): normal dist를 따르는 3 x 4 array 생성

np.random.standard_normal((3,4)): std normal dist. 괄호 하나 추가된거에 주목.

np.random.random((3,4)): 0~1의 uniform dist. 역시 tuple형태

np.append

- ✓ np.append([1,2,3],[[4,5,6],[7,8,9]])
- ✓ np.append([[1,2,3],[4,5,6]],[[7,8,9]],axis=0)
- ✓ np.append([[1,2,3],[4,5,6]],[[7],[8]],axis=1)

➔ 만약 np.append([[1,2,3],[4,5,6]],[7,8,9],axis=0)를 시도하면 에러난다. 합치는 두 array의 dimension이 같지 않기 때문.

np.concatenate(([1,2], [3,4]))

➔ 합치고 싶은 값들을 전부 괄호로 묶어서 튜플 형태로 만들어줘야 한다. 이게 append와의 차이점.

np.random.seed(0)