**NumPy**

* Object nparrays
* Create

np.zeros((3,3), dtype) # create 3\*3 matrix with zeros- **2D array**

np.ones(5) # create vector of length 5 with ones- **1D array**

np.eye(4) # create identity matrix of size 4 (square matrix) – **2D array**

np.full((2,3,4),5) # create 3-dims array 2 slis, 3 rows and 4 cols with 5 – **3D array**

np.array([[1,2],[3,4],[4,5]]) # create 3\*2 matrix, by a list of lists

np.arange(1,51) # create a sequence of numbers

* by create random variables

np.random.random((3,3)) # create 3\*3 matrix with **uniform(0,1)** entries

np.random.rand(100) # create 1D array filled with 100 uniform(0,1) entries

np.random.normal(0,1,(3,3)) # create 3\*3 matrix with standard normal entries

np.random.randn(100) # generate a 1D array filled with 100 random values

np.random.randint(10, size = (3,4,5)) # fill an array with random integers in range(10)

np.random.randint(1,10,10) # create an array

* by extract entries from old array

new\_array [ : : p+1] = x # in\_class\_p4

new\_array [1:n-1, 1:m-1] = 0 # hw5\_p1

* by concatenate two arrays

np.concatenate([A[np.newaxis, :, :], B[np.newaxis, :, :]], axis = 0) # in\_class\_p4

* attributes of nparrays

nparray.ndim

nparray.shape

nparray.size

nparray.dtype

* indexing

y[1,2,3]

* slicing **注意!**

# slice is a view of the original array, modifying the slice will modify the array

z = np.array(range(24)).reshape(3,4,2) # higher dimensional array

slice = z[1, :2, ] # slicing a subarray

ind\_slice = **slice.copy()** # make an independent copy

ind\_slice[0,0] = 90 # modifying the slice independently

* Concatenate

np.concatenate([A, B]) # concatenate two arrays (default along rows)

np.concatenate([A, B], axis = 1) # along columns

np.vstack([A, B]) # **vertical stack**: concatenate along rows

np.hstack([A, B]) # **horizontal stack**: concatenate along columns

* Split

left, right = np.split(x,[1], axis = 1) # split off at the first column

top, bottom = np.vsplit(x,[1]) # **vertical split**: split off at the first row

left, right = np.hsplit(x,[1]) # **horizontal split**: split off at the first column

* Data Aggregation

目的 to produce summary statistics

**重点**: Operations on nparrays are implemented as **vectorized functions**.

# Vectorized programs can run multiple operations from a single instruction, which can be carried out much faster than loop, doing operations on each element one-by-one.

np.sum | np.mean | np.std | np.var

np.min | np.max | np.median | np.percentile

np.argmin | np.argmax # find index of min and max values

* Broadcasting

解释原因It is possible to combine two arrays that are of different (but compatible) dimensions, but what if they are not in compatible dimensions? We need Broadcasting.

# Tip: To combine incompatible 不兼容dimensions arrays, we could “**inflate” NumPy arrays** by **np.newaxis** to map them to a higher dimension while occupying one “slice” of the array.

X = np.array([1, 2, 3]) X[ : , np.newaxis].shape **# (3,1)**

Y = np.array([0.1, 0.2]) Y[ np.newaxis, : ].shape **# (1,2)**

X[ : , np.newaxis] + Y[np.newaxis, : ]

* Iterating over an array

np.array ( [row.sum( ) for row in X] )

np.array ( [column.sum( ) for column in **X.T**] )

* Boolean Masks

# Boolean index, then apply aggregating functions

* np.sum (x<3, axis = 1) # count satisfying entries per row

# Extract a subset by Boolean index

* x[ (x<=34)&(x>=21)&(x%2!=0) ]

# return Boolean

* np.any(x<5) / np.all(x<5)
* Fancy Indexing

# to extract more than one observation at a time by directly specifying which observations you want

x[[0,2,4]] # extract the 1st, 3rd, 5th elements

x[1,[3,0]] # extract the 4th and 1st elements in the second row

* subset = np.array ([[0,1],[8,9]])
* x[subset]

* Sorting

# NumPy algorithms is more effective for NumPy arrays than other algorithms from base package.

# sorts every row of X separately.

# Values who were in the same column before may no longer be in the same column.

np.sort ( )

**np.argsort ( )**

* K-nearest neighbor procedure: Classification

# In a nutshell: You have a number of n-dimensional data values of different types (classes).

# To predict the type (class) of a new point, you find the k nearest neighbors of the point and take a

majority vote of their types.

* Data Visualization: matplotlib
* Broadcast the arrays.
* Find Euclidean distances between any two points.
* Find the k nearest neighbors for each point.

**Pandas**

\* Numpy array – list; stores only data of one type

\* Pandas series/data frame – dictionary; heterogeneous data types，**values + index**

Objects

* Series: 1D-array pd.Series { index : data } dict = { key : values }
* Data Frame: 2D-array
* Index immutable arrays or ordered sets

Creating Series

* # create from a list
* pd.Series ( [0.25, 0.5, 0.75, 1.0], index = [‘u’,’v’,’x’,’y’] ) # index is also a list
* # create from a dictionary
* pd.Series ( {‘A’: 210, ‘B’:49, ‘C’: 50} )

Creating Data Frames

* From a dictionary of series
* pd.DataFrame ({‘Treatment’: T, ‘Sections’: sections}) 参考课件
* From a list of dictionaries
* pd.DataFrame ( [{‘a’:1, ‘b’:2}, {‘b’:3, ‘c’:4}] )
* From a NumPy array
* pd.DataFrame (np.random.randint(3,2), columns=[‘foo’,’bar’], index = [‘a’,’b’,’c’])

Index operations

* Creating
* pd.Index ([2, 3, 4, 7, 11])

Set operations: & | ^

* Slicing
* data[‘a’:’c’] # slicing
* Masking (subset)
* data[ (data>0.3) & (data<0.9) ] # masking
* data [[‘a’, ‘d’]] # like fancy indexing in NumPy
* loc & iloc
* data.loc [2] # refer to the index you specified
* data.iloc [2] # refer to the internal python index
* classes.loc [ classes.Sections > 1, [‘Average Enrollment’, ‘Sections’]]

Data wrangling, cleaning and preparing:

* Read data from an external source
* Find which values are missing and what to do with them
* Find and correcting obvious mistakes
* Clean up elements you don’t want
* Combine/merge several data sets from different sources
* Restricting data sets
* Determine whether there are outliers
* Create summary statistics at one or more variables (also correlation)
* Produce graphs

From messy data to tidy data, more conductive for data analysis.

A screenshot of a cell phone

Description automatically generated

Tidy data, each variable forms a column, each observation forms a row, each observational unit forms a table

‘melting’ the data frame, means to ‘stack’ the two treatment variables

* melted\_df = pd.melt ( df1, id\_vars = ‘ ’, var\_name = ‘ ‘, value\_name = ‘ ’)

Merging Data Frames

df4 = pd.merge (melted\_df, Info, on = ‘Name’)

A screenshot of a cell phone

Description automatically generated

Grouping and Multi-Indexing

* By groupby:

# In groupby approach, aggregating data over crossed factors is suitable

* new = df4.groupby ([‘Treatment’, ‘Gender’])
* new.groups
* new.Result.mean ( ) # data aggregation
* By multi-Indexing:

# In multi-indexing approach, returns df4 as hierarchical indexed object

* df4.set\_index ([‘Treatment’, ‘Gender’],inplace = True) #’Treatment’ as the outside top-level variable
* df4.sort\_index (inplace = True) # sort a multi-index object
* df4.index
* df4.loc [(‘Treatment A’)] # extract all values for treatment A
* df4.loc [(‘Treatment A’)].Age.mean( )
* df4.loc [(‘Treatment A’, ‘F’)] # indexing in a multi-index setting
* df4.iloc [2:4] # slicing in a multi-index setting - rows
* df4.iloc [[0,1]] # slicing - rows

A screenshot of a cell phone

Description automatically generated

Question: so what is the difference between group-by method and multi-index method?

* Combine data frames
* pd.concat ([df1, df2]) # concatenate row
* pd.concat ([df3, df4], axis = 1) # concatenate along columns
* pd.concat ([df3, df4], ignore\_index = True)

Matplotib

Two different ways to make plots with matplotlib:

**【1】import pyplot as plt**

plt.plot ( ) # line plots

plt.scatter ( ) # scatter plots

plt.hist ( ) # histograms

* plt.plot (x, np.sin(x),color)

plt.xlim (10,0) # modify the axes

plt.ylim (1.2, -1.2)

plt.title (“A Sine Curve”, fontsize = 16) # label the plot

plt.xlabel (“x”)

plt.ylabel (“sin(x)”)

plt.plot (x, np.sin(x), label = ‘sin(x)’) # legend

plt.plot (x, np.cos(x), label = ‘cos(x)’)

plt.legend ( )

**【2】object oriented programming: fig / ax**

ax.plot ( )

ax.scatter ( )

ax.hist ( )

* Step 1. Create the empty figure environment

fig = plt.figure ( ) # create an empty figure container

ax = plt.axes ( ) # create axes object

* Step 2: Specify the range of axis and the function of plot

x = np.linspace (0,10,100) # 1-dim Numpy array

ax.plot (x, np.sin(x)) # plot

ax.set\_xlim (10,0) # modify the axes

ax.set\_ylim (1.2, -1.2)

ax.set\_title ( ) # label the plot

ax.set\_xlabel ( )

ax.set\_ylabel ( )

ax.plot (x, np.sin(x), label = ‘sin(x)’)

ax.plot (x, np.cos(x), label = ‘cos(x)’)

ax.legend ( )

better scatter plots

case study example:

post-opt 有没有延期毕业的可能性