

Data Analytics with pandas

Outline

- **pandas** Library
- **Series** Object
 - **Index** object
 - Vectorized operations
- **DataFrame** Object
 - Column/row operations
 - Loading data from CSV and Excel files
 - Subsetting with relational and logical operators
 - Splitting with group by and aggregation



pandas Library



- Most popular Python data analytics toolkit
 - Built on top of NumPy vectorized operations
- Extremely flexible data structures
 - Reading/writing from/to numerous data sources (CSV, JSON, ...)
 - Splitting, combining, merging, reshaping, joining data
- Fast computations with Series & DataFrames
 - Subsetting, summarizing, pivot tables, ...
 - Tightly integrated with **matplotlib** and **datetime** libraries
- Not part of standard Python installation
 - OS command prompt> **pip install pandas**
 - **LectPD_Cust_Loans.py**
 - >>> import pandas as pd**

Series Object



- An ordered, 1-dimensional list of data with an index
 - Unlike lists, each **series** element must be of the same data type

```
>>> rates_list = [0.07, 0.075, 0.07, 0.065, 0.077]
```

```
>>> rates = pd.Series(rates_list)
```

- Displaying the series

```
>>> rates
```

```
>>> rates.values # NumPy array
```

```
>>> rates.values.tolist() # Original list
```

```
>>> rates.index # RangeIndex object
```

```
>>> rates.index.tolist() # [0, 1, 2, 3, 4]
```

```
>>> rates
0    0.070
1    0.075
2    0.070
3    0.065
4    0.077
dtype: float64
```

Index Object



- Index object
 - Used to reference a single or multiple series elements
 - Most often a consecutive list of integers 0, 1, 2, ...
 - Could be any list of unique values – think dictionary keys

```
>>> loan_types
1022    Mortg
1023    Mortg
1024    Mortg
1025      Car
1026      Car
dtype: object
```

- Index derived from a list of loan ID's

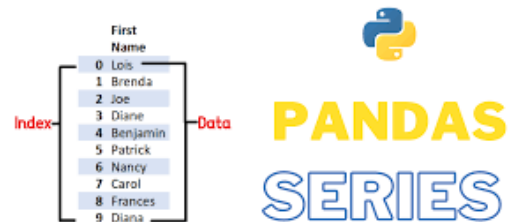
```
>>> loan_type_list = ['Mortg', 'Mortg', 'Mortg', 'Car', 'Car']
>>> loan_ids = list(range(1022, 1027))
>>> loan_types = pd.Series(loan_type_list, index=loan_ids)
```

- Series from dictionary → key becomes index

```
>>> amt_dict = {1022: 200000, 1023: 150000,
                1024: 100000, 1025: 25000, 1026: 10000}
>>> amounts = pd.Series(amt_dict)
```

Vectorized Operations

- Addition, multiplication, ...
- Index object must be lined up / aligned
 - Cannot multiply rates and amounts as is
- Redefine rates with loan IDs as indices
- Typically best to leave default indices unless there is a compelling reason to change them

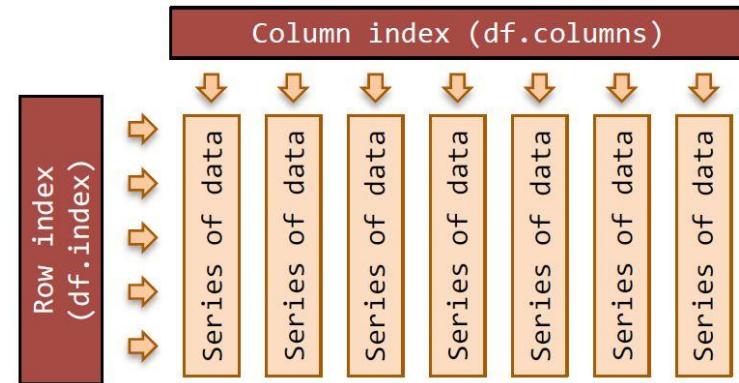


```
>>> rates * amounts
0      NaN
1      NaN
2      NaN
3      NaN
4      NaN
1022   NaN
1023   NaN
1024   NaN
1025   NaN
1026   NaN
dtype: float64
```

```
>>> rates * amounts
1022    14000.0
1023    11250.0
1024     7000.0
1025     1625.0
1026      770.0
dtype: float64
```

DataFrame Object

- Two-dimensional table of data
 - Columns represent attributes or characteristics of entities
 - Created from a distinct set of Series objects
 - Rows represent different instances of these entities
 - Both columns and rows indexed



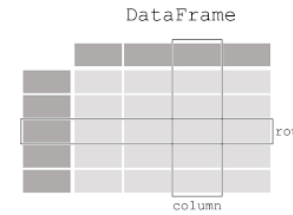
- Loans data frame

- See **loans.py**
- Assemble individual series into a list
- Create data frame with **concat**

```
>>> loans_df
      0      1      2      3      4
0  1022  0.070  Mortg  200000  15
1  1023  0.075  Mortg  150000  15
2  1024  0.070  Mortg  100000  30
3  1025  0.065   Car    25000   3
4  1026  0.077   Car    10000   5
```

```
>>> loan_series = loans.loan_series()
```

```
>>> loans_df = pd.concat(loan_series, axis=1)
```



DataFrame Function

- Redefine column indices as attribute names

```
>>> loan_cols = ['loanID', 'intRate', ..., 'loanTerm']
```

- Creating data frame from dict of individual series

```
>>> loan_series_dict = dict(zip(loan_cols, loan_series))
```

- Use **DataFrame** function to create the same data frame object

```
>>> loans_df = pd.DataFrame(loan_series_dict)
```

- Examine data frame contents

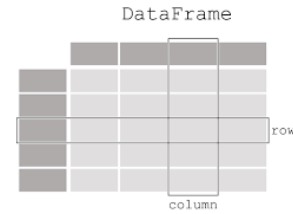
```
>>> loans_df.info()
```

- Basic column summary stats

```
>>> loans_df.describe()
```

```
>>> loans_df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
#   Column      Non-Null Count  Dtype
---  -
0   loanID      5 non-null      int64
1   intRate     5 non-null      float64
2   loanType    5 non-null      object
3   amount      5 non-null      int64
4   loanTerm    5 non-null      int64
dtypes: float64(1), int64(3), object(1)
memory usage: 328.0+ bytes
```


Working with DataFrame Columns



- Selecting a single column by name

```
>>> loans_df['intRate']
```

- Selecting multiple columns using names

- Must assemble column names into a list

```
>>> loans_df[['loanType', 'amount']]
```

- Selecting a column by number

- loanTerm is the 5th column at index 4

```
>>> loans_df[loans_df.columns[4]]
```

- Selecting multiple columns by slicing

- First 3 columns (0, 1, and 2)

```
>>> loans_df[loans_df.columns[:3]]
```

- Selecting non-adjacent columns

```
>>> loans_df[loans_df.columns[[1, 2, 4]]]
```



Working with DataFrame Rows

- Use index slicing when possible

- Select a single row

```
>>> loans_df[1:2]
```

- Select first 3 rows

```
>>> loans_df[:3]
```

- Select the last 2 rows using negative indexing

```
>>> loans_df[-2:]
```



- Use **loc** when slicing not possible

- Select non-consecutive rows

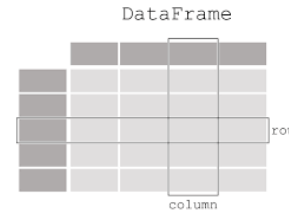
```
>>> loans_df.loc[[1, 3, 4]]
```

- Select rows based on logical (Boolean) column expressions

- Select all mortgage loans

```
>>> loans_df[loans_df['loanType']=='Mortg']
```

Subsetting with Logical Expressions



- Must review logical and relational operators
 - Logical AND operator & (amp symbol)
 - Multiple relational comparisons must be in ()
 - Loans with intRate 7% or higher with amounts over 100K
- ```
>>> loans_df[(loans_df['intRate']>=0.07) &
 (loans_df['amount']>100000)]
```

| !x    | Not x   |
|-------|---------|
| x   y | x OR y  |
| x & y | x AND y |

- Combining multiple relational and logical operators
  - Loans with rates 7% or higher who's terms are either less than 10 or greater than 20 years

```
>>> loans_df[(loans_df['intRate']>=0.07) &
 ((loans_df['loanTerm']<10) |
 (loans_df['loanTerm']>20))]
```

|    |                          |
|----|--------------------------|
| <  | less than                |
| <= | less than or equal to    |
| >  | greater than             |
| >= | greater than or equal to |
| == | exactly equal to         |
| != | not equal to             |

- Subsetting rows and columns
  - Mortgage loans, omitting loanType column

```
>>> loans_df.loc[loans_df['loanType']=='Mortg',
 loans_df.columns != 'loanType']
```

# Reading from CSV and Excel Files

- Read from CSV file into panda's data frame

```
>>> loans_df = pd.read_csv('Loans.csv')
>>> loans_df.info()
```

- Read from Excel file into panda's data frame

- Need to install **xlrd** library first

```
>>> loans_df = pd.read_excel('Loans.xls')
>>> loans_df.head()
>>> loans_df.tail()
```

- Other sources (not covered in class)

- Relational Database Management Systems (RDBMS)
- Java Script Object Notation (JSON) files
- Scraping Web pages for HTML tables, etc..



# More Subsetting Examples

- Easiest column subsetting is to create a list of quoted names
  - Mortgage loans showing loanID, amount, rate, term and payment

```
>>> loans_df.loc[loans_df['loanType']=='Mortg',
 ['loanID', 'amount', 'intRate', 'loanTerm', 'mthPmt']]
```

- More complex column subsetting with exclusion operator ~ (tilda symbol) and **isin** function

- Mortgage loans excluding first and last name and loan type

```
>>> loans_df.loc[loans_df['loanType']=='Mortg',
 ~loans_df.columns.isin(['firstName', 'lastName', 'loanType'])]
```

- The use of **!=** relational operator

- Showing all loans from customers that are not from Taos

```
>>> loans_no_Taos_df = loans_df.loc[loans_df['city']!='Taos', ...]
```

# More Subsetting Examples (cont.)

- Using relational and logical operators in complex expressions
  - Show all Taos mortgage loans

```
>>> loans_Taos_Mortg_df = loans_df.loc[(loans_df['city']=='Taos')
 & (loans_df['loanType']=='Mortg'),...]
```

- Working with both AND (&) and OR (|) logical operators
  - Mortgage loans either over half a million or under 200K

```
>>> loans_mortg_high_low_df = loans_df.loc[(loans_df['loanType']=='Mortg')
 & ((loans_df['amount']>500000) | (loans_df['amount']<200000)),...]
```

- Working with datetimes

- Issues with **loanDate** column

- NOT recognized as **datetime** from CSV file
- IS recognized as **datetime** from Excel file

- Showing all January loans (see **LectPD\_Cust\_Loans.py** for details)

```
>>> loans_jan_df = loans_df.loc[(loans_df['loanDate']>=beg_jan) &
 (loans_df['loanDate']<=end_jan), ...]
```

# Sorting and Adding Columns

- Sorting on one column
  - Provide the column to sort on and the sort order

```
>>> loans_sort1_df = loans_sub_df.sort_values('mthPmt',
 ascending=False)
```
- Sorting on multiple columns
  - Create a list of columns, first one must be categorical and sort order tuple

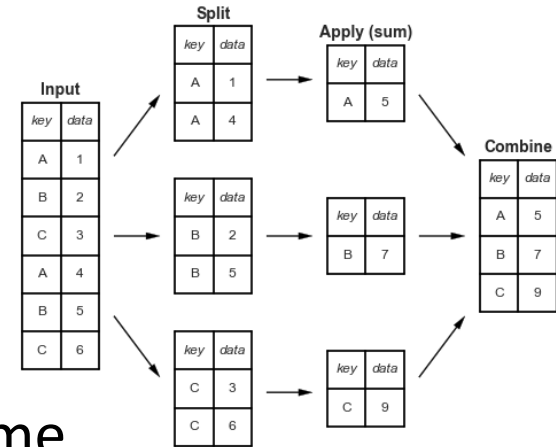
```
>>> loans_sort2_df = loans_sub_df.sort_values(['loanType',
 'amount'], ascending=(True, False))
```
- Creating a new column
  - Provide a name and an expression involving existing columns

```
>>> loans_df['totPmt'] = loans_df['mthPmt'] *
 loans_df['loanTerm'] * 12
```

# Group By (Splitting) and Aggregating

- Split -> Apply (Aggregate) -> Combine
- Split loans into 3 groups by loan type
  - Mortgage, Car and Other loans
- Apply aggregate operation on each group and combine the result into a new data frame

```
>>> loans_df.groupby('loanType')
```



- Average monthly payment

```
>>> loans_df.groupby('loanType').mean()['mthPmt']
```

- Total amount borrowed

```
>>> loans_df.groupby('loanType').sum()['amount']
```

- Number of loans by type

```
>>> loans_df.groupby('loanType').count()['loanID']
```



# Grouping By Multiple Columns

- Combine two (or more) categorical variables

- Find the average monthly payment by loan type and city

```
>>> loans_df.groupby(['loanType', 'city']).mean()['mthPmt']
```

- Use multiple aggregations

- Find the min, max and average monthly payment by loan type

- One element dictionary with column name as a key and aggregate operations as a list of values

```
>>> loans_df.groupby('loanType').agg({'mthPmt': ['min', 'max', 'mean']})
```

- Group by multiple columns and perform multiple aggregations

- Find the number of loans; total and average of amounts borrowed; min, max and average of monthly payments by loan type and city

- Three element dictionary with three column names as keys, followed by lists of operations on those columns as values

```
>>> agg_dict = {'loanID': ['count'], 'amount': ['sum', 'mean'],
 'mthPmt': ['min', 'max', 'mean']}
```

```
>>> loans_df.groupby(['loanType', 'city']).agg(agg_dict)
```

# Summary



- Introduced **pandas** library for data analysis
- Defined **Series** object
  - Supports vectorized operations using **Index** object
- Defined **DataFrame** object
  - The most important data structure for doing data analytics with pandas
  - Columns consists of series objects; rows represent different observations (instances) of various entities
  - Demonstrated how to work with columns and rows
- Loaded data frames from CSV and Excel files
  - Presented a variety of **subsetting** and **summarizing** operations with **relational** and **logical** operators
  - Showed how to **sort** on one or more columns
  - Finished with Split-Apply-Combine operations using **groupby** and aggregation functions