Week 10: Data Wrangling in Python

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

Overview

- Numerical analysis in Python
- Tabular data
- Pandas object types
- Working with dataframes in pandas
- Data input and output

Numerical Analysis in Python

Built-in Facilities

- As opposed to other programming languages (Julia, R, MatLab),
  Python provides very bare bones functionality for numeric analysis.
- E.g. no built-in matrix/array object type, limited mathematical and statistical functions.

```
# Representing 3x3 matrix with list
mat = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]]
```

```
# Subsetting 2nd row, 3rd element
mat[1][2]
```

```
6
```

```
# Naturally, this representation
# breaks down rather quickly
mat * 2
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

NumPy

- NumPy (**Num**eric **Py**thon) package provides the basis of numerical computing in Python:
  - multidimensional array (tensor);
  - mathematical functions for arrays;
  - array data I/O;
  - linear algebra, RNG, FFT, …

```
1 # Using 'as' allows to avoid typing full name
2 # each time the module is referred to
3 import numpy as np
```

NumPy Array

- Multidimensional array object is a principal container for data in Python:
  - Can be used to represent vectors, matrices and higher-dimensional tensors.
  - Can store images, audio, text, etc.
  - Forms the backbone of data frames in pandas.
- Number of dimensions (**axes**) depends on stored data:
  - E.g. vector of numbers is 1D, colour video is 4D (T x H x W x 3).
- Arrays are homogeneous and mutable.

```
1  vec = np.array([1, 2, 3])
2  vec
```

```
array([1, 2, 3])
```

```
1  arr = np.array([[1, 2, 3],
2                  [4, 5, 6],
3                  [7, 8, 9]])
```

NumPy Array: Example

```
1 arr
```

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

```
1 arr[1][2]
```

```
6
```

```
1 arr * 2
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

```
1 # Object type
2 type(arr)
```

```
<class 'numpy.ndarray'>
```

```
1 # Array dimensionality
2 arr.ndim
```

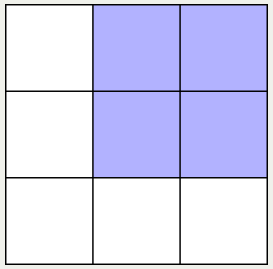
```
2
```

```
1 # Array size
2 arr.shape
```

```
(3, 3)
```

```
1 # Calculating summary statistics on array
2 # axis indicates the dimension
3 # compare to R's `apply(arr, 1, mean)`
4 # note that every list within a list
```

Array Indexing and Slicing

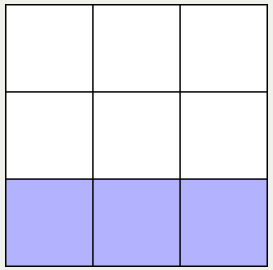


Expression

```
arr[:2, 1:]
```

Shape

```
(2, 2)
```

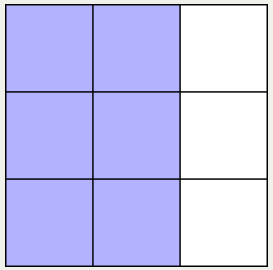


Expression

```
arr[2]
arr[2, :]
arr[2:, :]
```

Shape

```
(3,)
(3,)
(1, 3)
```



Expression

```
arr[:, :2]
```

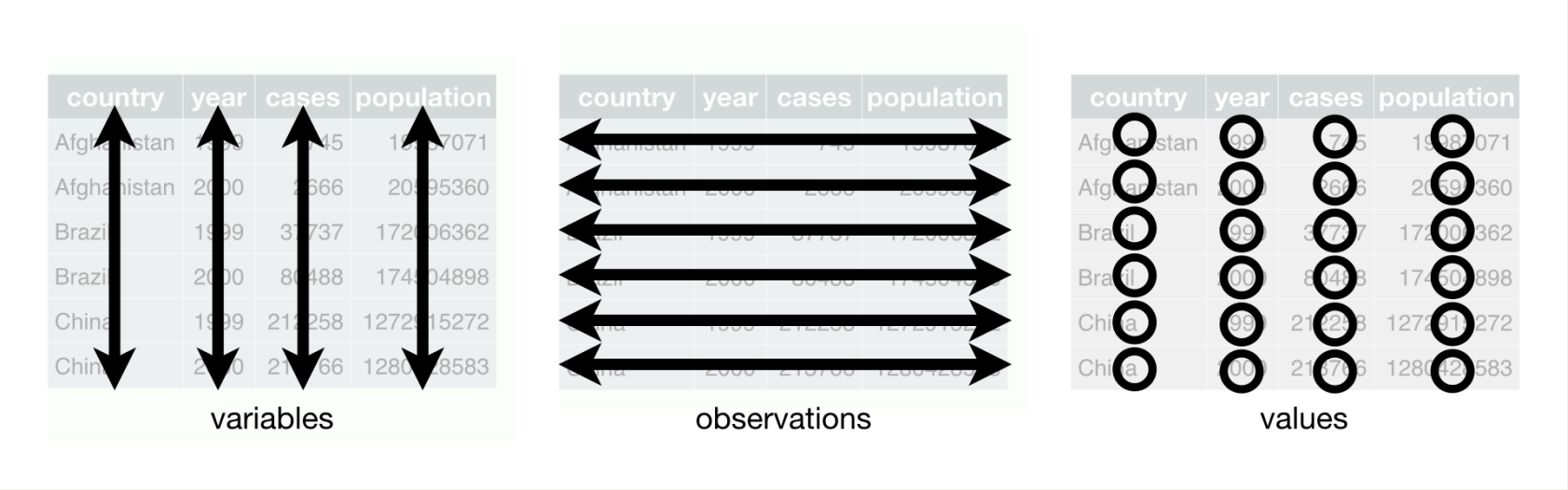
Shape

```
(3, 2)
```

Rectangular Data

Tidy Data

- Tidy data is a specific subset of rectangular data, where:
  - Each variable is in a column
  - Each observation is in a row
  - Each value is in a cell



Source

R for Data Science

Pandas

- Standard Python library does not have data type for rectangular data.
- However, `pandas` library has become the de facto standard for data manipulation.
- `pandas` is built upon (and often used in conjunction with) other computational libraries:
  - E.g. `scipy` (linear algebra), `statsmodels` (statistical models) and `scikit-learn` (machine learning)

```
1 import pandas as pd
```

Core `pandas` Object Types

- **Series** - one-dimensional sequence of values with an index.
- **DataFrame** - (typically) two-dimensional rectangular table of data.
- The key difference from NumPy arrays is that DataFrame can contain columns with different types.

Series

- **Series** is a one-dimensional array-like object.
- It contains a sequence of values and an associated array of data labels, called **index**.

```
1 ser = pd.Series([150.0, 120.0, 3000.0])
2 ser
```

```
0     150.0
1     120.0
2    3000.0
dtype: float64
```

```
1 ser.index # Retrieve index attribute
```

```
RangeIndex(start=0, stop=3, step=1)
```

```
1 ser.dtype # Retrieve data type attribute
```

```
dtype('float64')
```

```
1 ser[:2] # Slicing is similar to standard Python objects
```

```
0    150.0
1    120.0
dtype: float64
```

```
1 ser[ser > 200] # But subsetting with masks is also available
```

```
2    3000.0
dtype: float64
```

Indexing in Series

- Another way to think about Series is as a ordered dictionary.

```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

```
# Alternatively, we could have constructed this Series:
# ser = pd.Series(
#   [150.0, 120.0, 3000.0],
#   index=['apple', 'banana', 'watermelon']
# )
ser = pd.Series(d)
ser
```

```
apple          150.0
banana         120.0
watermelon    3000.0
dtype: float64
```

```
ser[:2] # This slicing would be impossible for standard dictionary
```

```
apple     150.0
banana    120.0
dtype: float64
```

```
ser[['apple', 'banana']] # But we can also use labels
```

```
apple     150.0
banana    120.0
dtype: float64
```

```
ser.index # Sequence of labels is converted into an Index object
```

```
Index(['apple', 'banana', 'watermelon'], dtype='object')
```

DataFrame

- **DataFrame** is a rectangular table of data.
- It is a workhorse of data analysis in `pandas`.

```
# DataFrame can be constructed from
# a dict of equal-length lists/arrays
data = {
  'fruit': ['apple', 'banana', 'watermelon'],
  'weight': [150.0, 120.0, 3000.0],
  'berry': [False, True, True]
}
df = pd.DataFrame(data)
df
```

```
        fruit  weight  berry
0       apple   150.0  False
1      banana   120.0   True
2  watermelon  3000.0   True
```

Working with Data Frames

Indexing in DataFrame

- DataFrame has both row (axis 0) and column (axis 1) indices.
- `DataFrame.iloc()` provides method for *index* location
- `DataFrame.loc()` provides method for *label* location

```
1 df.iloc[0] # First row
```

```
fruit     apple
weight    150.0
berry     False
Name: 0, dtype: object
```

```
1 df.iloc[:, 0] # First column by index
```

```
0         apple
1        banana
2    watermelon
Name: fruit, dtype: object
```

```
1 df.loc[:, 'fruit'] # First column by label
```

```
0         apple
1        banana
2    watermelon
Name: fruit, dtype: object
```

Summary of Indexing in DataFrame

Expression	Selection Operation
`df[val]`	Column or sequence of columns +convenience (e.g. slice)
`df.loc[lab_i]`	Row or subset of rows by label
`df.loc[:, lab_j]`	Column or subset of columns by label
`df.loc[lab_i, lab_j]`	Both rows and columns by label
`df.iloc[i]`	Row or subset of rows by integer position
`df.iloc[:, j]`	Column or subset of columns by integer position
`df.iloc[i, j]`	Both rows and columns by integer position
`df.at[lab_i, lab_j]`	Single scalar value by row and column label
`df.iat[i, j]`	Single scalar value by row and column integer position

Extra

Pandas documentation on indexing

Subsetting in DataFrame

```
1 df.iloc[:2] # Select the first two rows (with convenience shortcut for slic
```

```
    fruit  weight  berry
0   apple   150.0  False
1  banana   120.0   True
```

```
1 df[:2]  # Shortcut
```

```
    fruit  weight  berry
0   apple   150.0  False
1  banana   120.0   True
```

```
1 df.loc[:, ['fruit', 'berry']] # Select the columns 'fruit' and 'berry'
```

```
        fruit  berry
0       apple  False
1      banana   True
2  watermelon   True
```

```
1 df[['fruit', 'berry']] # Shortcut
```

```
        fruit  berry
0       apple  False
1      banana   True
2  watermelon   True
```

Columns in DataFrame

```
df.columns # Retrieve the names of all columns (index object)
```

```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

```
df.columns[0] # This Index object is subsettable
```

```
'fruit'
```

```
df.columns.str.startswith('fr') # As column names are strings, we can apply
```

```
array([ True, False, False])
```

```
df.iloc[:,df.columns.str.startswith('fr')] # This is helpful with more comp
```

```
        fruit
0       apple
1      banana
2  watermelon
```

Filtering in DataFrame

```
1 # Select rows where fruits are not berries
2 df[df.loc[:,'berry'] == False]
```

```
   fruit  weight  berry
0  apple   150.0  False
```

```
1 # The same can be achieved with more concise syntax
2 df[df['berry'] == False]
```

```
   fruit  weight  berry
0  apple   150.0  False
```

```
1 # Create new dataset with rows where weight is higher than 200
2 weight200 = df[df['weight'] > 200]
3 weight200
```

```
        fruit  weight  berry
2  watermelon  3000.0   True
```

Manipulating Columns in DataFrame

```
# Columns can be renamed with dictionary mapping
df = df.rename(columns = {'weight': 'weight_g'})
df
```

```
        fruit  weight_g  berry
0       apple     150.0  False
1      banana     120.0   True
2  watermelon    3000.0   True
```

```
# Columns can be added or modified by assignment
df['weight_oz'] = 0
df
```

```
        fruit  weight_g  berry  weight_oz
0       apple     150.0  False          0
1      banana     120.0   True          0
2  watermelon    3000.0   True          0
```

```
df['weight_oz'] = df['weight_g'] * 0.04
df
```

```
        fruit  weight_g  berry  weight_oz
0       apple     150.0  False        6.0
1      banana     120.0   True        4.8
2  watermelon    3000.0   True      120.0
```

```
df['weight_oz'] = round(df['weight_oz'], 1)
df
```

```
        fruit  weight_g  berry  weight_oz
0       apple     150.0  False        6.0
1      banana     120.0   True        4.8
2  watermelon    3000.0   True      120.0
```

Variable Transformation

- Lambda functions can be used to transform data with `map()` method

```
1 df['fruit'].map(lambda x: x.upper())
```

```
0          APPLE
1         BANANA
2     WATERMELON
Name: fruit, dtype: object
```

```
1 transform = lambda x: x.capitalize()
```

```
1 transformed = df['fruit'].map(transform)
```

```
1 transformed
```

```
0          Apple
1         Banana
2     Watermelon
Name: fruit, dtype: object
```

Piping in pandas

- While there is no special syntax (like %>%/|> in R), `pandas` provides a `pipe()` method to chain operations.

```
df = pd.DataFrame({
  'fruit': ['apple', 'banana', 'watermelon'],
  'weight': [150.0, 120.0, 3000.0],
  'berry': [False, True, True]
})
```

```
df = (
  df
  .pipe(lambda x: x.rename(columns = {'weight': 'weight_g'}))
  .pipe(lambda x: x.assign(
    fruit = x['fruit'].map(lambda x: x.capitalize()),
    weight_oz = round(x['weight_g'] * 0.04, 1))
  )
)
df
```

```
        fruit  weight_g  berry  weight_oz
0       Apple     150.0  False        6.0
1      Banana     120.0   True        4.8
2  Watermelon    3000.0   True      120.0
```

Data I/O

Data Formats

Format	Readability	Platform	Speed	Compression	Persistence
`csv`	✅ Human-readable	✅ Cross-platform	❌ Slow	❌ No	✅ Long-term
`rds`	❌ Binary	❌ R only	✅ Fast	✅ Yes	✅ Long-term
`pickle`	❌ Binary	❌ Python only	✅ Fast	✅ Yes	✅ Long-term
`parquet`	❌ Binary	✅ Cross-platform	✅ Fast	✅ Yes	✅ Long-term
`feather`	❌ Binary	✅ Cross-platform	✅ Fast	✅ Yes	❌ Short-term

File Object

- File object in Python provides the main interface to external files.
- In contrast to other core types, file objects are created not with a literal, but with a function, `open()`:

```
<variable_name> = open(<filepath>, <mode>)
```

```
1  # Create a new file object in write mode
2  f = open('../temp/test.txt', 'w')
```

Data Input and Output

- Modes of file objects allow to:
  - (`r`)ead a file (default);
  - (`w`)rite an object to a file;
  - e(`x`)clusively create, failing if a file exists;
  - (`a`)ppend to a file;
- `r+` mode if you need to read and write to file.

Data Output: Example

```
# Create a new file object in write mode
f = open('../temp/test.txt', 'w')
```

```
# Write a string of characters to it
f.write('This is a test file.')
```

```
20
```

```
# Flush output buffers to disk and close the connection
f.close()
```

Data Input: Example

- To avoid keeping track of open file connections, `with` statement can be used:

```
# Note that we use 'r' mode for reading
with open('../temp/test.txt', 'r') as f:
    text = f.read()
```

```
text
```

```
'This is a test file.'
```

> **Extra**
>
> Python documentation on with statement

Data I/O in `pandas`

- `pandas` provides high-level methods that takes care of file connections.
- These methods all follow the same `read_<format>` and `to_<format>` name patterns.
- CSV (comma-separated value) files are the standard of interoperability.

```
<variable_name> = pd.read_<format>(<filepath>)
```

```
<variable_name>.to_<format>(<filepath>)
```

```
1 df.to_csv('../temp/df.csv', index = False)
```

```
1 df = pd.read_csv('../temp/df.csv')
2 df
```

```
        fruit  weight_g  berry  weight_oz
0       Apple     150.0  False        6.0
1      Banana     120.0   True        4.8
2  Watermelon    3000.0   True      120.0
```

Real Data: Example

- We will use the data from Kaggle 2022 Machine Learning and Data Science Survey
- For more information you can read the executive summary
- Or explore the winning Python Jupyter Notebooks

```
# We specify that we want to combine first two rows as a header
kaggle2022 = pd.read_csv(
    '../data/kaggle_survey_2022_responses.csv',
    header = [0,1]
)
```

Visual Data Inspection

```
1 kaggle2022.head() # Returns the top n (n=5 default) rows
```

```
  Duration (in seconds)  ...
Q44_12
  Duration (in seconds)  ... Who/what are your favorite media sources that report on data science topics? (Select
all that apply) - Selected Choice - Other
0                   121  ...                                                  NaN
1                   462  ...                                                  NaN
2                   293  ...                                                  NaN
3                   851  ...                                                  NaN
4                   232  ...                                                  NaN

[5 rows x 296 columns]
```

```
1 kaggle2022.tail() # Returns the bottom n (n=5 default) rows
```

```
      Duration (in seconds)  ...
Q44_12
      Duration (in seconds)  ... Who/what are your favorite media sources that report on data science topics?
(Select all that apply) - Selected Choice - Other
23992                   331  ...                                                  NaN
23993                   330  ...                                                  NaN
23994                   860  ...                                                  NaN
23995                   597  ...                                                  NaN
23996                   303  ...                                                Other

[5 rows x 296 columns]
```

Data Summary: Numeric Variables

- DataFrame methods in pandas automatically handle (exclude) missing data (`NaN`)

```
# DataFrame.describe() provides an range of summary statistics
kaggle2022.describe()
```

```
      Duration (in seconds)  ...
Q44_11
      Duration (in seconds)  ... Who/what are your favorite media sources that report on data science topics?
(Select all that apply) - Selected Choice - None
count          2.399700e+04  ...                                                0.0
mean           1.009010e+04  ...                                                NaN
std            1.115403e+05  ...                                                NaN
min            1.200000e+02  ...                                                NaN
25%            2.640000e+02  ...                                                NaN
50%            4.140000e+02  ...                                                NaN
75%            7.150000e+02  ...                                                NaN
max            2.533678e+06  ...                                                NaN

[8 rows x 17 columns]
```

```
# Rather than using describe(),
# we can apply individual methods
kaggle2022.iloc[:,0].mean()
```

```
10090.095845313997
```

```
kaggle2022.iloc[:,0].median() # Median
```

```
414.0
```

```
kaggle2022.iloc[:,0].std() # Standard deviation
```

```
111540.30746801202
```

```
## We don't have to rely only on methods provided by `pandas`
import statistics
statistics.stdev(kaggle2022.iloc[:,0])
```

```
111540.30746801202
```

Data Summary: Categorical Variables

```
# Adding include = 'all' tells pandas to summarize all variables
kaggle2022.describe(include = 'all')
```

```
       Duration (in seconds)  ...
Q44_12
       Duration (in seconds)  ... Who/what are your favorite media sources that report on data science topics? (Select all that apply) - Selected Choice - Other
count           2.399700e+04  ...                                                  835
unique                   NaN  ...                                                    1
top                      NaN  ...                                                Other
freq                     NaN  ...                                                  835
mean            1.009010e+04  ...                                                  NaN
std             1.115403e+05  ...                                                  NaN
min             1.200000e+02  ...                                                  NaN
25%             2.640000e+02  ...                                                  NaN
50%             4.140000e+02  ...                                                  NaN
75%             7.150000e+02  ...                                                  NaN
max             2.533678e+06  ...                                                  NaN

[11 rows x 296 columns]
```

```
# Mode, most frequent value
kaggle2022.iloc[:,2].mode()
```

```
0    Man
Name: (Q3, What is your gender? - Selected Choice), dtype: object
```

Data Summary: Tabulation

```
1 # Counts of unique values
2 kaggle2022.iloc[:,2].value_counts()
```

```
(Q3, What is your gender? - Selected Choice)
Man                          18266
Woman                         5286
Prefer not to say              334
Nonbinary                       78
Prefer to self-describe         33
Name: count, dtype: int64
```

```
1 # We can further normalize them by the number of rows
2 kaggle2022.iloc[:,2].value_counts(normalize = True)
```

```
(Q3, What is your gender? - Selected Choice)
Man                          0.761178
Woman                        0.220278
Prefer not to say            0.013918
Nonbinary                    0.003250
Prefer to self-describe      0.001375
Name: proportion, dtype: float64
```

Descriptive Statistics Methods

Method	Numeric	Categorical	Description
`count`	yes	yes	Number of non-NA observations
`value_counts`	yes	yes	Number of unique observations by value
`describe`	yes	yes	Set of summary statistics for Series/DataFrame
`min`, `max`	yes	yes (caution)	Minimum and maximum values
`quantile`	yes	no	Sample quantile ranging from 0 to 1
`sum`	yes	yes (caution)	Sum of values
`prod`	yes	no	Product of values
`mean`	yes	no	Mean
`median`	yes	no	Median (50% quantile)
`var`	yes	no	Sample variance
`std`	yes	no	Sample standard deviation
`skew`	yes	no	Sample skewness (third moment)
`kurt`	yes	no	Sample kurtosis (fourth moment)

Next

- Tutorial: Reading/writing and reshaping data in Python
- Next week: Classes and Object-oriented programming