Week 10: Data Wrangling in Python

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

14 November 2022

Module website: tinyurl.com/POP77001

Overview

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

Numerical analysis in Python

- 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

Numerical analysis in Python

- 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

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

Numerical analysis in Python

- 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

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

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

Out[2]: 6

Numerical analysis in Python

- 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

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

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

Out[2]:
```
6
```

In [3]:
```
# Naturally, this representation
# breaks down rather quickly
mat * 2
```

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

NumPy - numerical analysis in Python

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

NumPy - numerical analysis in Python

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

In [4]:
```
# Using 'as' allows to avoid typing full name
# each time the module is referred to
import numpy as np
```

NumPy array

- Multidimensional (N) array object (aka ndarray) is a principal container for datasets in Python.
- It is the backbone of data frames, operating behind the scenes

NumPy array

- Multidimensional (N) array object (aka ndarray) is a principal container for datasets in Python.
- It is the backbone of data frames, operating behind the scenes

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

NumPy array

- Multidimensional (N) array object (aka ndarray) is a principal container for datasets in Python.
- It is the backbone of data frames, operating behind the scenes

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

In [6]:
```
arr[1][2]
```

Out[6]: 6

NumPy array

- Multidimensional (N) array object (aka ndarray) is a principal container for datasets in Python.
- It is the backbone of data frames, operating behind the scenes

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

In [6]:
```
arr[1][2]
```

Out[6]:
```
6
```

In [7]:
```
arr * 2
```

Out[7]:
```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

Working with arrays

Working with arrays

In [8]:
```
# Object type
type(arr)
```

Out[8]:
```
numpy.ndarray
```

Working with arrays

In [8]:
```
# Object type
type(arr)
```

Out[8]:
```
numpy.ndarray
```

In [9]:
```
# Array dimensionality
arr.ndim
```

Out[9]:
```
2
```

Working with arrays

In [8]:
```
# Object type
type(arr)
```

Out[8]:
```
numpy.ndarray
```

In [9]:
```
# Array dimensionality
arr.ndim
```

Out[9]:
```
2
```

In [10]:
```
# Array size
arr.shape
```

Out[10]:
```
(3, 3)
```

Working with arrays

In [8]:
```
# Object type
type(arr)
```

Out[8]:
```
numpy.ndarray
```

In [9]:
```
# Array dimensionality
arr.ndim
```

Out[9]:
```
2
```

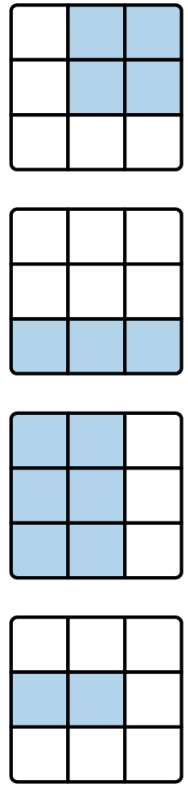
In [10]:
```
# Array size
arr.shape
```

Out[10]:
```
(3, 3)
```

In [11]:
```
# Calculating summary statistics on array
# axis indicates the dimension
# compare to R's `apply(arr, 1, mean)`
# note that every list within a list
# is treated as a column (not row)
arr.mean(axis = 0)
```

Out[11]:
```
array([4., 5., 6.])
```

Array indexing and slicing

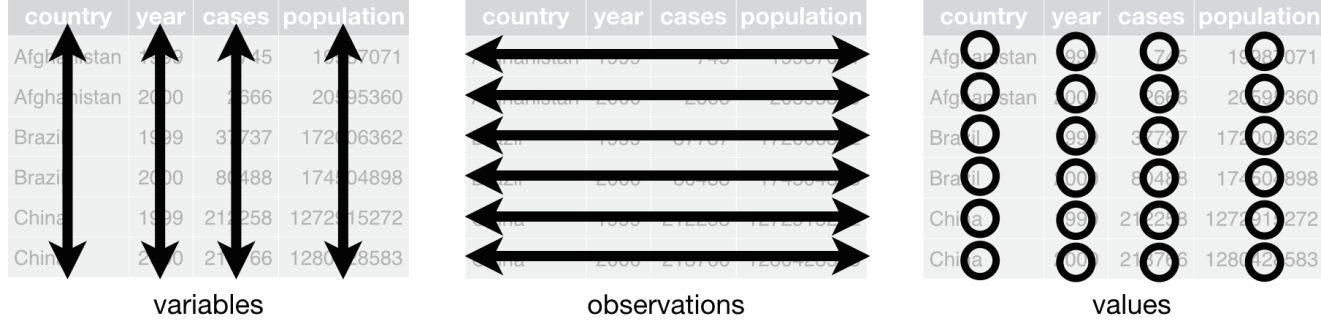


Expression	Shape
`arr[:2,1:]`	`(2,2)`
`arr[2]`	`(3,)`
`arr[2, :]`	`(3,)`
`arr[2:, :]`	`(1,3)`
`arr[:, :2]`	`(3,2)`
`arr[1, :2]`	`(2,)`
`arr[1:2, :2]`	`(1,2)`

Source: Python for Data Analysis

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 conjuction with) other computational libraries
- E.g. `numpy` (array data type), `scipy` (linear algebra) and `scikit-learn` (machine learning)

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 conjuction with) other computational libraries
- E.g. `numpy` (array data type), `scipy` (linear algebra) and `scikit-learn` (machine learning)

In [12]:
```
import pandas as pd
```

Core `pandas` object types

- *Series* - one-dimensional sequence of values
- *DataFrame* - (typically) two-dimensional rectangular table

Series

- *Series* is a one-dimensional array-like object

Series

- *Series* is a one-dimensional array-like object

In [13]:
```
sr1 = pd.Series([150.0, 120.0, 3000.0])
sr1
```

Out[13]:
```
0     150.0
1     120.0
2    3000.0
dtype: float64
```

Series

- *Series* is a one-dimensional array-like object

In [13]:
```
sr1 = pd.Series([150.0, 120.0, 3000.0])
sr1
```

Out[13]:
```
0     150.0
1     120.0
2    3000.0
dtype: float64
```

In [14]:
```
sr1[0] # Slicing is similar to standard Python objects
```

Out[14]:
```
150.0
```

Series

- *Series* is a one-dimensional array-like object

In [13]:
```
sr1 = pd.Series([150.0, 120.0, 3000.0])
sr1
```

Out[13]:
```
0     150.0
1     120.0
2    3000.0
dtype: float64
```

In [14]:
```
sr1[0] # Slicing is similar to standard Python objects
```

Out[14]:
```
150.0
```

In [15]:
```
sr1[sr1 > 200] # But subsetting is also available
```

Out[15]:
```
2    3000.0
dtype: float64
```

Indexing in Series

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

Indexing in Series

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

In [16]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

Indexing in Series

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

In [16]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [17]:
```
sr2 = pd.Series(d)
sr2
```

Out[17]:
```
apple          150.0
banana         120.0
watermelon    3000.0
dtype: float64
```

Indexing in Series

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

In [16]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [17]:
```
sr2 = pd.Series(d)
sr2
```

Out[17]:
```
apple          150.0
banana         120.0
watermelon    3000.0
dtype: float64
```

In [18]:
```
sr2[0] # Recall that this slicing would be impossible for standard dict
```

Out[18]:
```
150.0
```

Indexing in Series

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

In [16]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [17]:
```
sr2 = pd.Series(d)
sr2
```

Out[17]:
```
apple          150.0
banana         120.0
watermelon    3000.0
dtype: float64
```

In [18]:
```
sr2[0] # Recall that this slicing would be impossible for standard dict
```

Out[18]:
```
150.0
```

In [19]:
```
sr2.index # Sequence of labels is converted into an Index object
```

Out[19]:
```
Index(['apple', 'banana', 'watermelon'], dtype='object')
```

DataFrame - the workhorse of data analysis

- *DataFrame* is a rectangular table of data

DataFrame - the workhorse of data analysis

- *DataFrame* is a rectangular table of data

In [20]:
```
data = {'fruit': ['apple', 'banana', 'watermelon'], # DataFrame can be
        'weight': [150.0, 120.0, 3000.0],           # a dict of equal-l
        'berry': [False, True, True]}
df = pd.DataFrame(data)
df
```

Out[20]:

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

Indexing in DataFrame

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

Indexing in DataFrame

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

In [21]:
```
df.iloc[0] # First row
```

Out[21]:
```
fruit     apple
weight    150.0
berry     False
Name: 0, dtype: object
```

Indexing in DataFrame

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

In [21]:
```
df.iloc[0] # First row
```

Out[21]:
```
fruit     apple
weight    150.0
berry     False
Name: 0, dtype: object
```

In [22]:
```
df.iloc[:,0] # First column
```

Out[22]:
```
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

Subsetting in DataFrame

In [23]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[23]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

Subsetting in DataFrame

In [23]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[23]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [24]:
```
df[:2]  # Shortcut
```

Out[24]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

Subsetting in DataFrame

In [23]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[23]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [24]:
```
df[:2]  # Shortcut
```

Out[24]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [25]:
```
df.loc[:, ['fruit', 'berry']] # Select the columns 'fruit' and 'berry'
```

Out[25]:

	fruit	berry
0	apple	False
1	banana	True
2	watermelon	True

Subsetting in DataFrame

In [23]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[23]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [24]:
```
df[:2]  # Shortcut
```

Out[24]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [25]:
```
df.loc[:, ['fruit', 'berry']] # Select the columns 'fruit' and 'berry'
```

Out[25]:

	fruit	berry
0	apple	False
1	banana	True
2	watermelon	True

In [26]:
```
df[['fruit', 'berry']] # Shortcut
```

Out[26]:

	fruit	berry
0	apple	False
1	banana	True
2	watermelon	True

Columns in DataFrame

Columns in DataFrame

In [27]:
```
df.columns # Retrieve the names of all columns (index object)
```

Out[27]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

Columns in DataFrame

In [27]:
```
df.columns # Retrieve the names of all columns (index object)
```

Out[27]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

In [28]:
```
df.columns[0] # This Index object is subsettable
```

Out[28]:
```
'fruit'
```

Columns in DataFrame

In [27]:
```
df.columns # Retrieve the names of all columns (index object)
```

Out[27]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

In [28]:
```
df.columns[0] # This Index object is subsettable
```

Out[28]:
```
'fruit'
```

In [29]:
```
df.columns.str.startswith('fr') # As column names are strings, we can a
```

Out[29]:
```
array([ True, False, False])
```

Columns in DataFrame

In [27]:
```
df.columns # Retrieve the names of all columns (index object)
```

Out[27]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

In [28]:
```
df.columns[0] # This Index object is subsettable
```

Out[28]:
```
'fruit'
```

In [29]:
```
df.columns.str.startswith('fr') # As column names are strings, we can a
```

Out[29]:
```
array([ True, False, False])
```

In [30]:
```
df.iloc[:,df.columns.str.startswith('fr')] # This is helpful with more
```

Out[30]:

	fruit
0	apple
1	banana
2	watermelon

Filtering in DataFrame

Filtering in DataFrame

In [31]:
```
# Select rows where fruits are not berries
df[df.loc[:,'berry'] == False]
```

Out[31]:

	fruit	weight	berry
0	apple	150.0	False

Filtering in DataFrame

In [31]:
```
# Select rows where fruits are not berries
df[df.loc[:,'berry'] == False]
```

Out[31]:

	fruit	weight	berry
0	apple	150.0	False

In [32]:
```
# The same can be achieved with more concise syntax
df[df['berry'] == False]
```

Out[32]:

	fruit	weight	berry
0	apple	150.0	False

Filtering in DataFrame

In [31]:
```
# Select rows where fruits are not berries
df[df.loc[:,'berry'] == False]
```

Out[31]:

	fruit	weight	berry
0	apple	150.0	False

In [32]:
```
# The same can be achieved with more concise syntax
df[df['berry'] == False]
```

Out[32]:

	fruit	weight	berry
0	apple	150.0	False

In [33]:
```
# Create new dataset with rows where weight is higher than 200
weight200 = df[df['weight'] > 200]
weight200
```

Out[33]:

	fruit	weight	berry
2	watermelon	3000.0	True

Manipulating columns in DataFrame

Manipulating columns in DataFrame

In [34]:
```
df = df.rename(columns = {'weight': 'weight_g'}) # Columns can be renam
```

Manipulating columns in DataFrame

In [34]:
```
df = df.rename(columns = {'weight': 'weight_g'}) # Columns can be renam
```

In [35]:
```
df
```

Out[35]:

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

Manipulating columns in DataFrame

In [34]:
```
df = df.rename(columns = {'weight': 'weight_g'}) # Columns can be renam
```

In [35]:
```
df
```

Out[35]:

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

In [36]:
```
df['weight_oz'] = 0  # Columns can be added or modified by assignment
```

Manipulating columns in DataFrame

In [34]:
```
df = df.rename(columns = {'weight': 'weight_g'}) # Columns can be renam
```

In [35]:
```
df
```

Out[35]:

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

In [36]:
```
df['weight_oz'] = 0  # Columns can be added or modified by assignment
```

In [37]:
```
df
```

Out[37]:

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

Manipulating columns in DataFrame continued

Manipulating columns in DataFrame continued

In [38]:
```
df['weight_oz'] = df['weight_g'] * 0.04
```

Manipulating columns in DataFrame continued

In [38]:
```
df['weight_oz'] = df['weight_g'] * 0.04
```

In [39]:
```
df
```

Out[39]:

	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

Manipulating columns in DataFrame continued

In [38]:
```
df['weight_oz'] = df['weight_g'] * 0.04
```

In [39]:
```
df
```

Out[39]:

	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

In [40]:
```
df['weight_oz'] = round(df['weight_oz'], 1)
```

Manipulating columns in DataFrame continued

In [38]:
```
df['weight_oz'] = df['weight_g'] * 0.04
```

In [39]:
```
df
```

Out[39]:

	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

In [40]:
```
df['weight_oz'] = round(df['weight_oz'], 1)
```

In [41]:
```
df
```

Out[41]:

	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

Variable transformation

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

In [42]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[42]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

Variable transformation

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

In [42]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[42]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

In [43]:
```
transform = lambda x: x.capitalize()
```

Variable transformation

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

In [42]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[42]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

In [43]:
```
transform = lambda x: x.capitalize()
```

In [44]:
```
transformed = df['fruit'].map(transform)
```

Variable transformation

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

In [42]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[42]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

In [43]:
```
transform = lambda x: x.capitalize()
```

In [44]:
```
transformed = df['fruit'].map(transform)
```

In [45]:
```
transformed
```

Out[45]:
```
0         Apple
1        Banana
2    Watermelon
Name: fruit, dtype: object
```

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>)
```

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
- You can `r+` mode if you need to read and write to file

Data output example

Data output example

In [46]:
```
f = open('../temp/test.txt', 'w') # Create a new file object in write m
```

Data output example

In [46]:
```
f = open('../temp/test.txt', 'w') # Create a new file object in write m
```

In [47]:
```
f.write('This is a test file.') # Write a string of characters to it
```

Out[47]:
```
20
```

Data output example

In [46]:
```
f = open('../temp/test.txt', 'w') # Create a new file object in write m
```

In [47]:
```
f.write('This is a test file.') # Write a string of characters to it
```

Out[47]:
```
20
```

In [48]:
```
f.close() # Flush output buffers to disk and close the connection
```

Data input example

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

Extra: Python documentation on with statement

Data input example

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

Extra: Python documentation on with statement

In [49]:
```
with open('../temp/test.txt', 'r') as f: # Note that we use 'r' mode fo
    text = f.read()
```

Data input example

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

Extra: Python documentation on with statement

In [49]:
```
with open('../temp/test.txt', 'r') as f: # Note that we use 'r' mode fo
    text = f.read()
```

In [50]:
```
text
```

Out[50]:
```
'This is a test file.'
```

Reading and writing data 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>)
```

Reading data in `pandas` example

- We will use the data from Kaggle 2021 Machine Learning and Data Science Survey
- For more information you can read the executive summary
- Or explore the winning Python Jupyter Notebooks

Reading data in `pandas` example

- We will use the data from Kaggle 2021 Machine Learning and Data Science Survey
- For more information you can read the executive summary
- Or explore the winning Python Jupyter Notebooks

In [51]:
```
# We specify that we want to combine first two rows as a header
kaggle2021 = pd.read_csv(
    '../data/kaggle_survey_2021_responses.csv',
    header = [0,1]
)
```

```
/tmp/ipykernel_215835/1877677354.py:2: DtypeWarning: Columns (1
95,201) have mixed types. Specify dtype option on import or set
low_memory=False.
  kaggle2021 = pd.read_csv(
```

Visual data inspection

Visual data inspection

In [52]:
```
kaggle2021.head() # Returns the top n (n=5 default) rows
```

Out[52]:

	Time from Start to Finish (seconds)	Q1	Q2	Q3	Q4	Q5	
	Duration (in seconds)	What is your age (# years)?	What is your gender? - Selected Choice	In which country do you currently reside?	What is the highest level of formal education that you have attained or plan to attain within the next 2 years?	Select the title most similar to your current role (or most recent title if retired): - Selected Choice	For hov yeai yo writin program
0	910	50-54	Man	India	Bachelor's degree	Other	5-1

1	784	50-54	Man	Indonesia	Master’s degree	Program/Project Manager	20
2	924	22-24	Man	Pakistan	Master’s degree	Software Engineer	1-
3	575	45-49	Man	Mexico	Doctoral degree	Research Scientist	20
4	781	45-49	Man	India	Doctoral degree	Other	<

5 rows × 369 columns

Visual data inspection continued

Visual data inspection continued

In [53]:
```
kaggle2021.tail() # Returns the bottom n (n=5 default) rows
```

Out[53]:

	Time from Start to Finish (seconds)	Q1	Q2	Q3	Q4	Q5	
	Duration (in seconds)	What is your age (# years)?	What is your gender? - Selected Choice	In which country do you currently reside?	What is the highest level of formal education that you have attained or plan to attain within the next 2 years?	Select the title most similar to your current role (or most recent title if retired): - Selected Choice	For how years you writing a programn
25968	1756	30-34	Man	Egypt	Bachelor's degree	Data Analyst	1-3

25969	253	22-24	Man	China	Master’s degree	Student	1-3
25970	494	50-54	Man	Sweden	Doctoral degree	Research Scientist	I have written
25971	277	45-49	Man	United States of America	Master’s degree	Data Scientist	5-10
25972	255	18-21	Man	India	Bachelor’s degree	Business Analyst	I have written

5 rows × 369 columns

Reading in other (non-`.csv`) data files

- Pandas can read in file other than `.csv` (comma-separated value)
- Common cases include STATA `.dta`, SPSS `.sav` and SAS `.sas`
- Use `pd.read_stata(path)`, `pd.read_spss(path)` and `pd.read_sas(path)`
- Check here for more examples

Writing data out in Python

- Note that when writing data out we start with the object name storing the dataset
- I.e. `df.to_csv(path)` as opposed to `df = pd.read_csv(path)`
- Pandas can also write out into other data formats
- E.g. `df.to_excel(path)`, `df.to_stata(path)`

Writing data out in Python

- Note that when writing data out we start with the object name storing the dataset
- I.e. `df.to_csv(path)` as opposed to `df = pd.read_csv(path)`
- Pandas can also write out into other data formats
- E.g. `df.to_excel(path)`, `df.to_stata(path)`

In [54]:
```
kaggle2021.to_csv('../temp/kaggle2021.csv')
```

Summarizing numeric variables

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

Summarizing numeric variables

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

In [55]:
```
kaggle2021.describe() # DataFrame.describe() provides an range of summa
```

Out[55]:

	Time from Start to Finish (seconds)	Q30_B_Part_1	Q30_B_Part_2	Q30_B_Part_3	Q30_B_Part

	Duration (in seconds)	In the next 2 years, do you hope to become more familiar with any of these specific data storage products? (Select all that apply) - Selected Choice - Microsoft Azure Data Lake Storage	In the next 2 years, do you hope to become more familiar with any of these specific data storage products? (Select all that apply) - Selected Choice - Microsoft Azure Disk Storage	In the next 2 years, do you hope to become more familiar with any of these specific data storage products? (Select all that apply) - Selected Choice - Amazon Simple Storage Service (S3)	In the nex years, do y hope become mo familiar w any of the specific da stora product (Select all th apply Select Choic Amaz Elastic F System (EF
count	2.597300e+04	0.0	0.0	0.0	0
mean	1.105466e+04	NaN	NaN	NaN	Na
std	1.014716e+05	NaN	NaN	NaN	Na
min	1.200000e+02	NaN	NaN	NaN	Na
25%	4.430000e+02	NaN	NaN	NaN	Na
50%	6.560000e+02	NaN	NaN	NaN	Na
75%	1.038000e+03	NaN	NaN	NaN	Na
max	2.488653e+06	NaN	NaN	NaN	Na

Methods for summarizing numeric variables

Methods for summarizing numeric variables

In [56]:
```
kaggle2021.iloc[:,0].mean() # Rather than using describe(), we can appl
```

Out[56]:
```
11054.66492126439
```

Methods for summarizing numeric variables

In [56]:
```
kaggle2021.iloc[:,0].mean() # Rather than using describe(), we can appl
```

Out[56]:
```
11054.66492126439
```

In [57]:
```
kaggle2021.iloc[:,0].median() # Median
```

Out[57]:
```
656.0
```

Methods for summarizing numeric variables

In [56]:
```
kaggle2021.iloc[:,0].mean() # Rather than using describe(), we can appl
```

Out[56]:
```
11054.66492126439
```

In [57]:
```
kaggle2021.iloc[:,0].median() # Median
```

Out[57]:
```
656.0
```

In [58]:
```
kaggle2021.iloc[:,0].std() # Standard deviation
```

Out[58]:
```
101471.6221245172
```

Methods for summarizing numeric variables

In [56]:
```
kaggle2021.iloc[:,0].mean() # Rather than using describe(), we can appl
```

Out[56]:
```
11054.66492126439
```

In [57]:
```
kaggle2021.iloc[:,0].median() # Median
```

Out[57]:
```
656.0
```

In [58]:
```
kaggle2021.iloc[:,0].std() # Standard deviation
```

Out[58]:
```
101471.6221245172
```

In [59]:
```
import statistics ## We don't have to rely only on methods provided by
statistics.stdev(kaggle2021.iloc[:,0])
```

Out[59]:
```
101471.6221245172
```

Summarizing categorical variables

Summarizing categorical variables

In [60]:
```
kaggle2021.describe(include = 'all') # Adding include = 'all' tells par
```

Out[60]:

	Time from Start to Finish (seconds)	Q1	Q2	Q3	Q4	Q5	
	Duration (in seconds)	What is your age (# years)?	What is your gender? - Selected Choice	In which country do you currently reside?	What is the highest level of formal education that you have attained or plan to attain within the next 2 years?	Select the title most similar to your current role (or most recent title if retired): - Selected Choice	For h y wri progr
count	2.597300e+04	25973	25973	25973	25973	25973	
unique	NaN	11	5	66	7	15	

top	NaN	25-29	Man	India	Master's degree	Student
freq	NaN	4931	20598	7434	10132	6804
mean	1.105466e+04	NaN	NaN	NaN	NaN	NaN
std	1.014716e+05	NaN	NaN	NaN	NaN	NaN
min	1.200000e+02	NaN	NaN	NaN	NaN	NaN
25%	4.430000e+02	NaN	NaN	NaN	NaN	NaN
50%	6.560000e+02	NaN	NaN	NaN	NaN	NaN
75%	1.038000e+03	NaN	NaN	NaN	NaN	NaN
max	2.488653e+06	NaN	NaN	NaN	NaN	NaN

11 rows × 369 columns

Methods for summarizing categorical variables

Methods for summarizing categorical variables

In [61]:
```
kaggle2021.iloc[:,2].mode() # Mode, most frequent value
```

Out[61]:
```
0    Man
Name: (Q2, What is your gender? - Selected Choice), dtype: obje
ct
```

Methods for summarizing categorical variables

In [61]:
```
kaggle2021.iloc[:,2].mode() # Mode, most frequent value
```

Out[61]:
```
0    Man
Name: (Q2, What is your gender? - Selected Choice), dtype: obje
ct
```

In [62]:
```
kaggle2021.iloc[:,2].value_counts() # Counts of unique values
```

Out[62]:
```
Man                            20598
Woman                           4890
Prefer not to say                355
Nonbinary                         88
Prefer to self-describe           42
Name: (Q2, What is your gender? - Selected Choice), dtype: int6
4
```

Methods for summarizing categorical variables

In [61]:
```
kaggle2021.iloc[:,2].mode() # Mode, most frequent value
```

Out[61]:
```
0    Man
Name: (Q2, What is your gender? - Selected Choice), dtype: obje
ct
```

In [62]:
```
kaggle2021.iloc[:,2].value_counts() # Counts of unique values
```

Out[62]:
```
Man                          20598
Woman                         4890
Prefer not to say              355
Nonbinary                       88
Prefer to self-describe         42
Name: (Q2, What is your gender? - Selected Choice), dtype: int6
4
```

In [63]:
```
kaggle2021.iloc[:,2].value_counts(normalize = True) # We can further nc
```

Out[63]:
```
Man                          0.793054
Woman                        0.188272
Prefer not to say            0.013668
Nonbinary                    0.003388
Prefer to self-describe      0.001617
```

Summary of 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