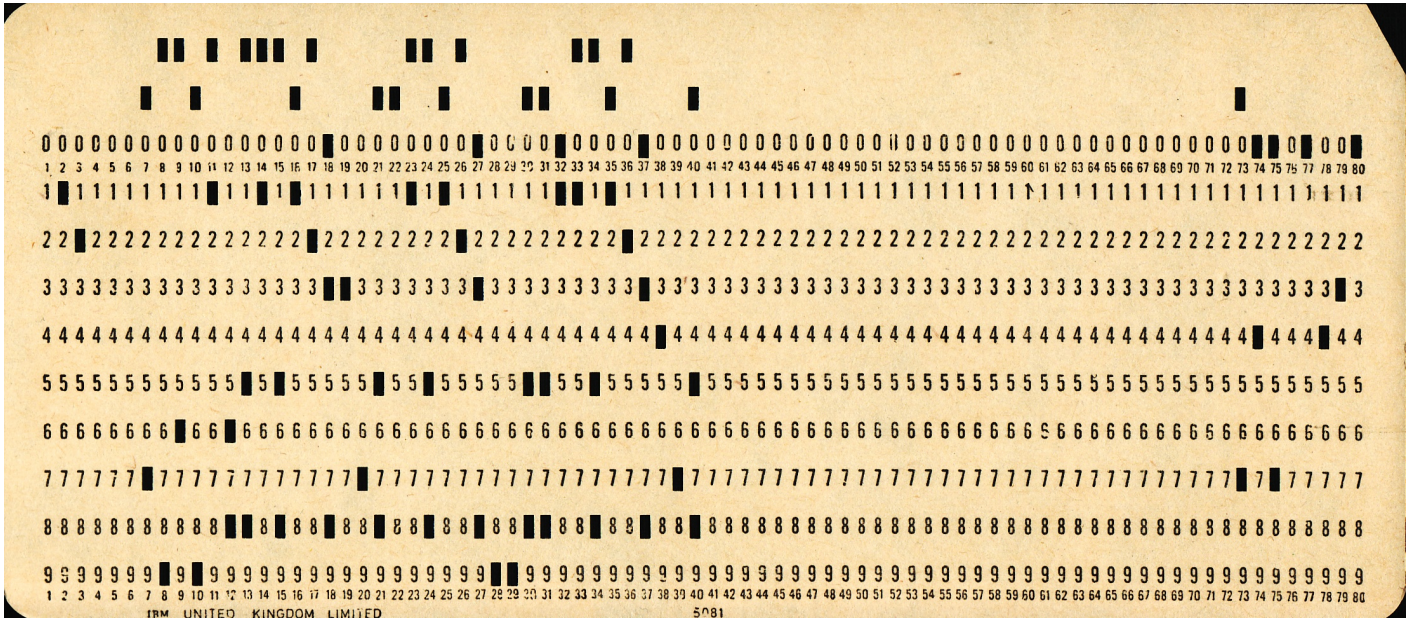
Day 1, Part 3: Pandas and Data I/O

Introduction to Python

Tom Paskhalis

RECSM Summer School 2023

Rectangular data

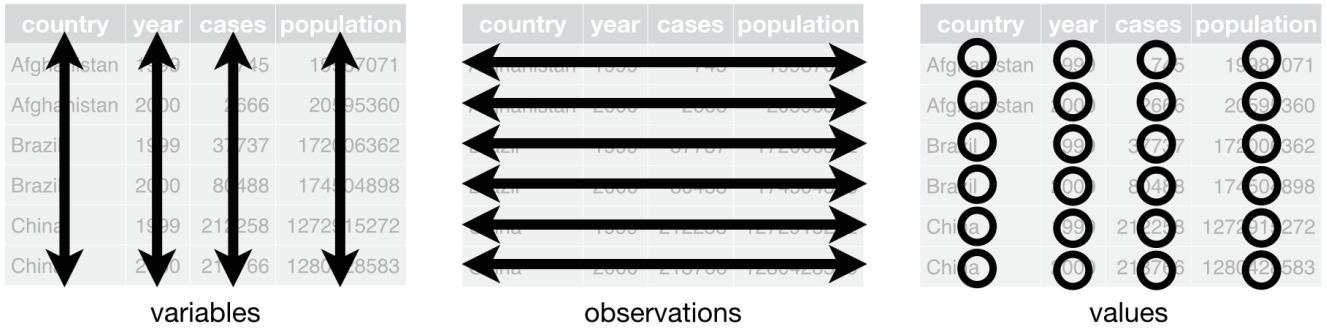


History of rectangular data goes back to punchcards with origins in US census data processing.

Source: Wikipedia

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

Data in Python

- Python can hold and manipulate > 1 dataset at the same time
- Python stores objects in memory
- The limit on the size of data is determined by your computer memory
- Most functionality for dealing with data is provided by external libraries

Pandas

- Standard Python library does not have data type for tabular 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 tabular 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 [1]:
```
# Using 'as' allows to avoid typing full name each time the module is r
import pandas as pd
```

Series

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

Series

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

In [2]:
```
sr1 = pd.Series([150.0, 120.0, 3000.0])
sr1
```

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

Series

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

In [2]:
```
sr1 = pd.Series([150.0, 120.0, 3000.0])
sr1
```

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

In [3]:
```
sr1[0] # Slicing is simiar to standard Python objects
```

Out[3]:
```
150.0
```

Series

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

In [2]:
```
sr1 = pd.Series([150.0, 120.0, 3000.0])
sr1
```

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

In [3]:
```
sr1[0] # Slicing is simiar to standard Python objects
```

Out[3]:
```
150.0
```

In [4]:
```
sr1[sr1 > 200]
```

Out[4]:
```
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 [5]:
```
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 [5]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [6]:
```
sr2 = pd.Series(d)
sr2
```

Out[6]:
```
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 [5]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [6]:
```
sr2 = pd.Series(d)
sr2
```

Out[6]:
```
apple          150.0
banana         120.0
watermelon    3000.0
dtype: float64
```

In [7]:
```
sr2[0] # Recall that this slicing would be impossible for standard dict
```

Out[7]:
```
150.0
```

Indexing in Series

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

In [5]:
```
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [6]:
```
sr2 = pd.Series(d)
sr2
```

Out[6]:
```
apple          150.0
banana         120.0
watermelon    3000.0
dtype: float64
```

In [7]:
```
sr2[0] # Recall that this slicing would be impossible for standard dict
```

Out[7]:
```
150.0
```

In [8]:
```
sr2.index
```

Out[8]:
```
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 [9]:
```
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[9]:

	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 [10]:
```
df.iloc[0] # First row
```

Out[10]:
```
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 [10]:
```
df.iloc[0] # First row
```

Out[10]:
```
fruit     apple
weight    150.0
berry     False
Name: 0, dtype: object
```

In [11]:
```
df.iloc[:,0] # First column
```

Out[11]:
```
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 [12]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[12]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

Subsetting in DataFrame

In [12]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[12]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [13]:
```
df[:2]  # Shortcut
```

Out[13]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

Subsetting in DataFrame

In [12]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[12]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [13]:
```
df[:2]  # Shortcut
```

Out[13]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [14]:
```
df.loc[:, ['fruit', 'berry']] # Select the columns 'fruit' and 'berry'
```

Out[14]:

	fruit	berry
0	apple	False
1	banana	True
2	watermelon	True

Subsetting in DataFrame

In [12]:
```
df.iloc[:2] # Select the first two rows (with convenience shortcut for
```

Out[12]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [13]:
```
df[:2]  # Shortcut
```

Out[13]:

	fruit	weight	berry
0	apple	150.0	False
1	banana	120.0	True

In [14]:
```
df.loc[:, ['fruit', 'berry']] # Select the columns 'fruit' and 'berry'
```

Out[14]:

	fruit	berry
0	apple	False
1	banana	True
2	watermelon	True

In [15]:
```
df[['fruit', 'berry']] # Shortcut
```

Out[15]:

	fruit	berry
0	apple	False
1	banana	True
2	watermelon	True

Columns in DataFrame

Columns in DataFrame

In [16]:
```
df.columns # Retrieve the names of all columns
```

Out[16]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

Columns in DataFrame

In [16]:
```
df.columns # Retrieve the names of all columns
```

Out[16]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

In [17]:
```
df.columns[0] # This Index object is subsettable
```

Out[17]:
```
'fruit'
```

Columns in DataFrame

In [16]:
```
df.columns # Retrieve the names of all columns
```

Out[16]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

In [17]:
```
df.columns[0] # This Index object is subsettable
```

Out[17]:
```
'fruit'
```

In [18]:
```
df.columns.str.startswith('fr') # As column names are strings, we can a
```

Out[18]:
```
array([ True, False, False])
```

Columns in DataFrame

In [16]:
```
df.columns # Retrieve the names of all columns
```

Out[16]:
```
Index(['fruit', 'weight', 'berry'], dtype='object')
```

In [17]:
```
df.columns[0] # This Index object is subsettable
```

Out[17]:
```
'fruit'
```

In [18]:
```
df.columns.str.startswith('fr') # As column names are strings, we can a
```

Out[18]:
```
array([ True, False, False])
```

In [19]:
```
df.iloc[:,df.columns.str.startswith('fr')] # This is helpful with more
```

Out[19]:

	fruit
0	apple
1	banana
2	watermelon

Filtering in DataFrame

Filtering in DataFrame

In [20]:
```
df[df.loc[:,'berry'] == False] # Select rows where fruits are not berri
```

Out[20]:

	fruit	weight	berry
0	apple	150.0	False

Filtering in DataFrame

In [20]:
```
df[df.loc[:,'berry'] == False] # Select rows where fruits are not berri
```

Out[20]:

	fruit	weight	berry
0	apple	150.0	False

In [21]:
```
df[df['berry'] == False] # The same can be achieved with more concise s
```

Out[21]:

	fruit	weight	berry
0	apple	150.0	False

Filtering in DataFrame

In [20]:
```
df[df.loc[:,'berry'] == False] # Select rows where fruits are not berri
```

Out[20]:

	fruit	weight	berry
0	apple	150.0	False

In [21]:
```
df[df['berry'] == False] # The same can be achieved with more concise s
```

Out[21]:

	fruit	weight	berry
0	apple	150.0	False

In [22]:
```
weight200 = df[df['weight'] > 200] # Create new dataset with rows where
weight200
```

Out[22]:

	fruit	weight	berry
2	watermelon	3000.0	True

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 [23]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[23]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

Variable transformation

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

In [23]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[23]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

In [24]:
```
transform = lambda x: x.capitalize()
```

Variable transformation

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

In [23]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[23]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

In [24]:
```
transform = lambda x: x.capitalize()
```

In [25]:
```
transformed = df['fruit'].map(transform)
```

Variable transformation

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

In [23]:
```
df['fruit'].map(lambda x: x.upper())
```

Out[23]:
```
0         APPLE
1        BANANA
2    WATERMELON
Name: fruit, dtype: object
```

In [24]:
```
transform = lambda x: x.capitalize()
```

In [25]:
```
transformed = df['fruit'].map(transform)
```

In [26]:
```
transformed
```

Out[26]:
```
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 [27]:
```
f = open('../temp/test.txt', 'w') # Create a new file object in write m
```

Data output example

In [27]:
```
f = open('../temp/test.txt', 'w') # Create a new file object in write m
```

In [28]:
```
f.write('This is a test file.') # Write a string of characters to it
```

Out[28]:
```
20
```

Data output example

In [27]:
```
f = open('../temp/test.txt', 'w') # Create a new file object in write m
```

In [28]:
```
f.write('This is a test file.') # Write a string of characters to it
```

Out[28]:
```
20
```

In [29]:
```
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 [30]:
```
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 [30]:
```
with open('../temp/test.txt', 'r') as f: # Note that we use 'r' mode fo
    text = f.read()
```

In [31]:
```
text
```

Out[31]:
```
'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 [32]:
```
# We specify that we want to combine first two rows as a header
kaggle2021 = pd.read_csv('../data/kaggle_survey_2021_responses.csv', he
```

```
/home/tpaskhalis/.local/lib/python3.8/site-packages/IPython/cor
e/interactiveshell.py:3441: DtypeWarning: Columns (195,201) have
mixed types.Specify dtype option on import or set low_memory=Fal
se.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

Visual data inspection

Visual data inspection

In [33]:
```
kaggle2021.head() # Returns the top n (n=5 default) rows
```

Out[33]:

	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 program
0	910	50-54	Man	India	Bachelor's	Other	5-10

					degree		
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
3	575	45-49	Man	Mexico	Doctoral degree	Research Scientist	20+
4	781	45-49	Man	India	Doctoral degree	Other	< 1

5 rows × 369 columns

Visual data inspection continued

Visual data inspection continued

In [34]:
```
kaggle2021.tail() # Returns the bottom n (n=5 default) rows
```

Out[34]:

	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 r years you writing ar programm
25968	1756	30-34	Man	Egypt	Bachelor's	Data	1-3 y

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

- 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 `pandas`

- 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 [35]:
```
kaggle2021.to_csv('../temp/kaggle2021.csv')
```

Additional pandas materials

Books:

- McKinney, Wes. 2022. *Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter*. 3rd ed. Sebastopol, CA: O'Reilly Media

  From the original author of the library!

Online:

- Pandas Getting Started Tutorials
- Pandas Documentation (intermediate and advanced)

Tomorrow

- Exploratory data analysis
- Data visualization