Week 10: Data Wrangling

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

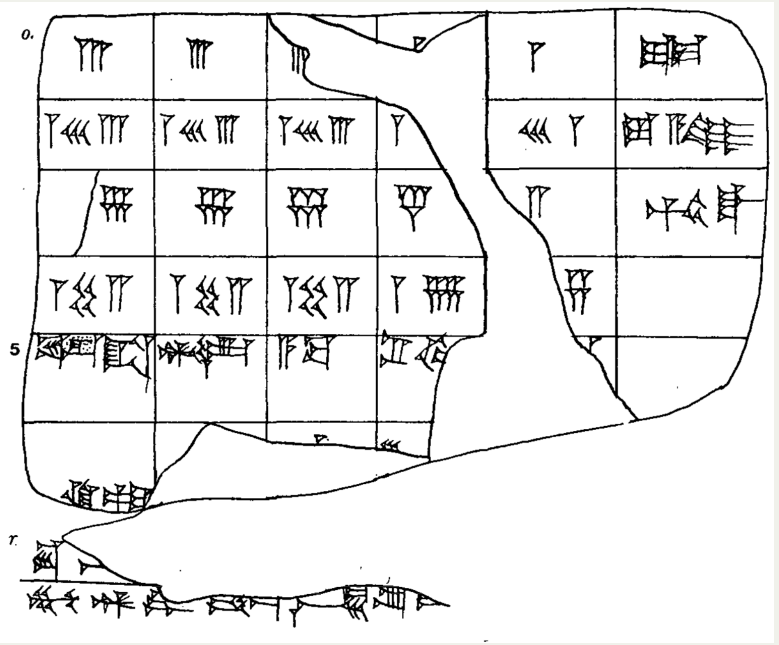
Overview

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

Rectangular Data

Older Tabular Data

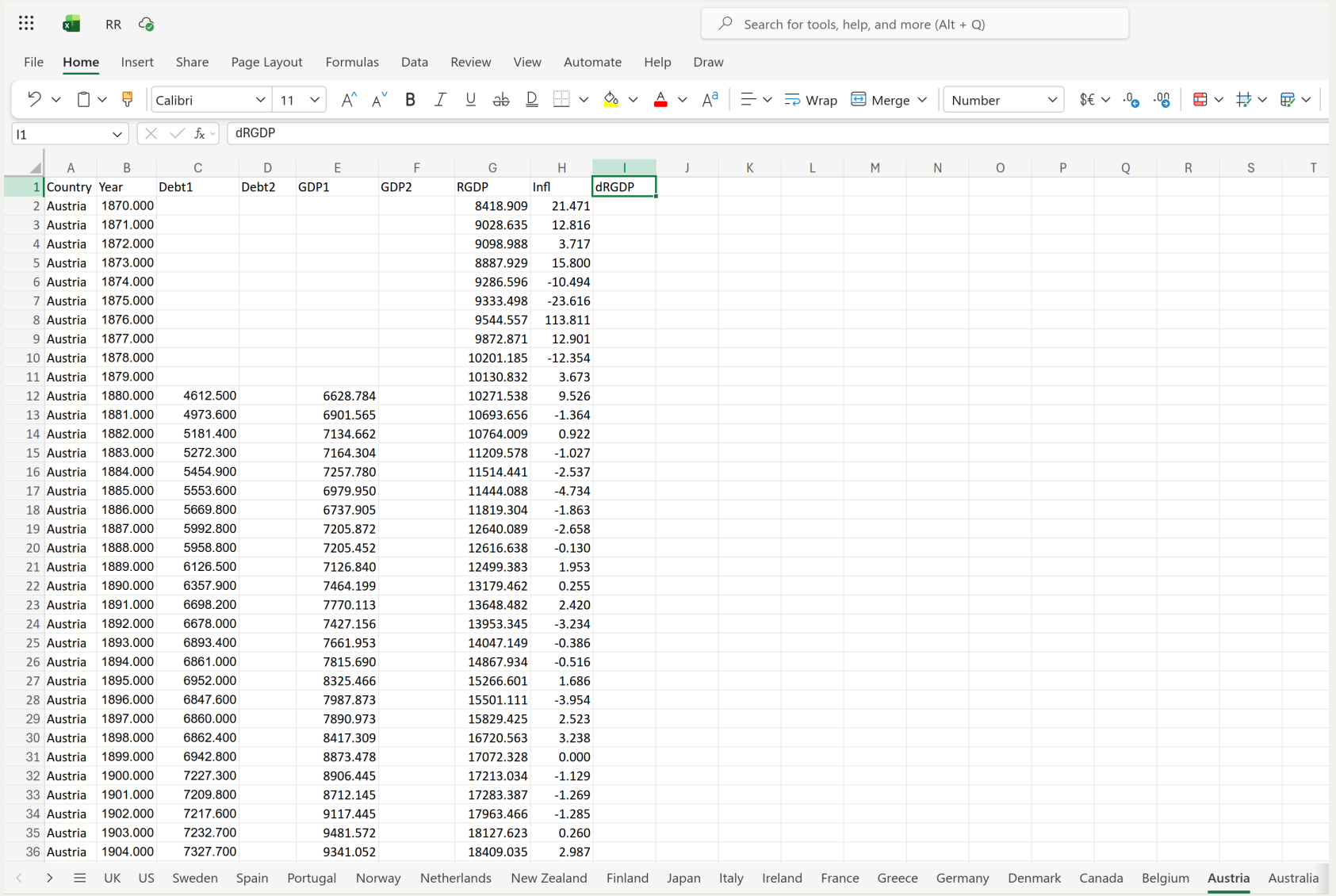
Earliest known (Robson, 2003) tabular account (Puzrish-Dagan, Ur, c.2028 BCE):



Source: (Sigrist, 1984)

3	3	3	2	1	lambs
93	93	93	6[2]	31	first-rate sheep
6	6	6	4	2	billy goats
102	102	102	68	[3]4	

Newer Tabular Data

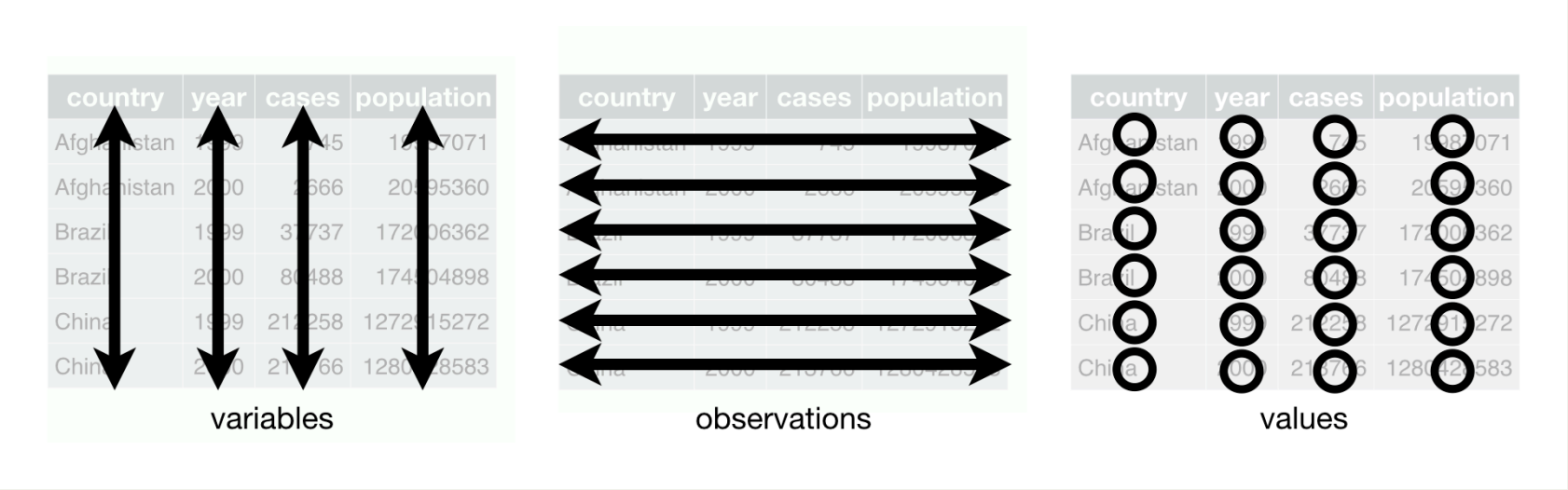


	A	B	C	D	E	F	G	H	I
1	Country	Year	Debt1	Debt2	GDP1	GDP2	RGDP	Infl	dRGDP
2	Austria	1870.000					8418.909	21.471	
3	Austria	1871.000					9028.635	12.816	
4	Austria	1872.000					9098.988	3.717	
5	Austria	1873.000					8887.929	15.800	
6	Austria	1874.000					9286.596	-10.494	
7	Austria	1875.000					9333.498	-23.616	
8	Austria	1876.000					9544.557	113.811	
9	Austria	1877.000					9872.871	12.901	
10	Austria	1878.000					10201.185	-12.354	
11	Austria	1879.000					10130.832	3.673	
12	Austria	1880.000	4612.500		6628.784		10271.538	9.526	
13	Austria	1881.000	4973.600		6901.565		10693.656	-1.364	
14	Austria	1882.000	5181.400		7134.662		10764.009	0.922	
15	Austria	1883.000	5272.300		7164.304		11209.578	-1.027	
16	Austria	1884.000	5454.900		7257.780		11514.441	-2.537	
17	Austria	1885.000	5553.600		6979.950		11444.088	-4.734	
18	Austria	1886.000	5669.800		6737.905		11819.304	-1.863	
19	Austria	1887.000	5992.800		7205.872		12640.089	-2.658	
20	Austria	1888.000	5958.800		7205.452		12616.638	-0.130	
21	Austria	1889.000	6126.500		7126.840		12499.383	1.953	
22	Austria	1890.000	6357.900		7464.199		13179.462	0.255	
23	Austria	1891.000	6698.200		7770.113		13648.482	2.420	
24	Austria	1892.000	6678.000		7427.156		13953.345	-3.234	
25	Austria	1893.000	6893.400		7661.953		14047.149	-0.386	
26	Austria	1894.000	6861.000		7815.690		14867.934	-0.516	
27	Austria	1895.000	6952.000		8325.466		15266.601	1.686	
28	Austria	1896.000	6847.600		7987.873		15501.111	-3.954	
29	Austria	1897.000	6860.000		7890.973		15829.425	2.523	
30	Austria	1898.000	6862.400		8417.309		16720.563	3.238	
31	Austria	1899.000	6942.800		8873.478		17072.328	0.000	
32	Austria	1900.000	7227.300		8906.445		17213.034	-1.129	
33	Austria	1901.000	7209.800		8712.145		17283.387	-1.269	
34	Austria	1902.000	7217.600		9117.445		17963.466	-1.285	
35	Austria	1903.000	7232.700		9481.572		18127.623	0.260	
36	Austria	1904.000	7327.700		9341.052		18409.035	2.987	

Source: (Reinhardt & Rogoff, 2010) and (Herndon, Ash and Pollin, 2018)

Tidy Data

- Tidy data (Wickham, 2014) is a specific subset of rectangular data, where:
  - Each variable forms a column
  - Each observation forms a row
  - Each value forms a cell



Extra

R for Data Science

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 C).
- 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]
```

```
np.int64(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
```

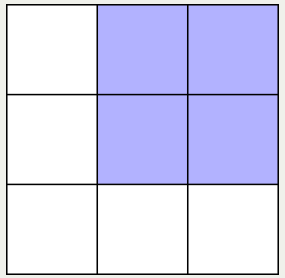
```
(3, 3)
```

```
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
```

```
# is treated as a column (not row)
arr.mean(axis = 0)
```

```
array([4., 5., 6.])
```

Array Indexing and Slicing

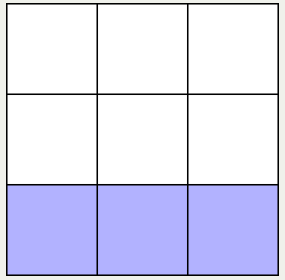


Expression

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

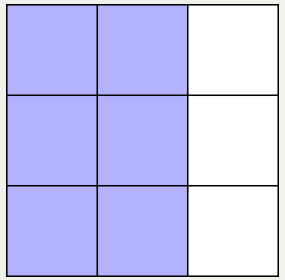
Shape

```
(2, 2)
```



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

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



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

```
(3, 2)
```

Data Wrangling in Python

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='str')
```

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
```

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: str
```

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

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

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 slicing)
```

```
    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

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

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

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

```
'fruit'
```

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

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

```
1 df.iloc[:,df.columns.str.startswith('fr')] # This is helpful with more complicated column selection criteria
```

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

Filtering in DataFrame

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

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

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

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

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

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

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

Data I/O in R

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

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 Wrangling in R

Data Frames

- Data frame is one of the object types available in base R.
- Despite their matrix-like appearance, data frames are lists of equal-sized vectors.
- Data frames can be created with `data.frame()` function with named vectors as input.

```r
df <- data.frame(
    x = 1:4,
    y = c("a", "b", "c", "d"),
    z = c(TRUE, FALSE, FALSE, TRUE)
)
df
```

```
  x y     z
1 1 a  TRUE
2 2 b FALSE
3 3 c FALSE
4 4 d  TRUE
```

Data Frame: Example

```r
# str() function applied to data frame is useful in determining variable types
str(df)
```

```
'data.frame':   4 obs. of  3 variables:
 $ x: int  1 2 3 4
 $ y: chr  "a" "b" "c" "d"
 $ z: logi  TRUE FALSE FALSE TRUE
```

```r
# dim() function behaves similar to matrix, showing N rows and N columns, respectively
dim(df)
```

```
[1] 4 3
```

```r
# Note that nrow() function is just a convenient wrapper for dim()[1L]
nrow
```

```
function (x)
dim(x)[1L]
<bytecode: 0x5676edf95068>
<environment: namespace:base>
```

```r
# Same with ncol() which is a wrapper for dim()[2L]
ncol
```

```
function (x)
dim(x)[2L]
<bytecode: 0x5676ed6b0348>
<environment: namespace:base>
```

```r
# In contrast to matrix length() of data frame displays the length of underlying list
length(df)
```

```
[1] 3
```

Creating Data Frame: Example

```r
l <- list(x = 1:5, y = letters[1:5], z = rep(c(TRUE, FALSE), length.out = 5))
l
```

```
$x
[1] 1 2 3 4 5

$y
[1] "a" "b" "c" "d" "e"

$z
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

```r
df <- data.frame(l)
df
```

```
  x y     z
1 1 a  TRUE
2 2 b FALSE
3 3 c  TRUE
4 4 d FALSE
5 5 e  TRUE
```

```r
str(df)
```

```
'data.frame':   5 obs. of  3 variables:
 $ x: int  1 2 3 4 5
 $ y: chr  "a" "b" "c" "d" ...
 $ z: logi  TRUE FALSE TRUE FALSE TRUE
```

Subsetting Data Frame

- In subsetting data frames the techniques of subsetting matrices and lists are combined:
  - If you subset with a single vector, it behaves as a list
  - If you subset with two vectors, it behaves as a matrix

```
data.frame[[index]]
data.frame[index]
data.frame$name
data.frame[vector_1, vector_2]
```

Subsetting Data Frame: Example

```r
# Like a list
df[["z"]]
```

```
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

```r
df["z"]
```

```
      z
1  TRUE
2 FALSE
3  TRUE
4 FALSE
5  TRUE
```

```r
df[c("x", "z")]
```

```
  x     z
1 1  TRUE
2 2 FALSE
3 3  TRUE
4 4 FALSE
5 5  TRUE
```

```r
# Like a matrix
df[,c("x", "z")]
```

```
  x     z
1 1  TRUE
2 2 FALSE
3 3  TRUE
4 4 FALSE
5 5  TRUE
```

```r
df[df$y == "b",]
```

```
  x y     z
2 2 b FALSE
```

Beyond Data Frames

Data Frame Limitations

- While very versatile (and available out-of-the-box) data frames have their drawbacks:
  - Individual cells (observations) cannot themselves be lists;
  - Somewhat limited (and inconsistent) data manipulation functions;
  - Memory inefficient (**copy-on-modify** semantics);
  - No parallelisation.

Data Frame Alternatives

- Two major alternatives to/enhanced versions of data frames are:
  - `tibble` from `tibble` package (part of `tidyverse` package ecosystem)
  - `data.table` from `data.table`
- `tibble` provides features enhancing user experience (readability, ease of manipulation)
- `data.table` provides speed

Data Table - Fast Data Frame

- As opposed to data frames, data tables are updated by reference.
- This frees up a lot of RAM for big data!
- It provides low-level parallelism.
- SQL-like operations for data manipulation.
- Has no external dependencies (other than base R itself).

```r
library("data.table")
```

```r
dt <- data.table::data.table(
    x = 1:4,
    y = c("a", "b", "c", "d"),
    z = c(TRUE, FALSE, FALSE, TRUE)
)
dt
```

```
      x      y      z
   <int> <char> <lgcl>
1:     1      a   TRUE
2:     2      b  FALSE
3:     3      c  FALSE
4:     4      d   TRUE
```

Data Table: Example

- The syntax of `data.table()` might be a bit confusing at first.

```
1 dt[,z]
```

```
[1]  TRUE FALSE FALSE  TRUE
```

```
1 dt[,.(z)]
```

```
       z
   <lgcl>
1:   TRUE
2:  FALSE
3:  FALSE
4:   TRUE
```

```
1 dt[,.(x, z)]
```

```
      x      z
   <int> <lgcl>
1:     1   TRUE
2:     2  FALSE
3:     3  FALSE
4:     4   TRUE
```

```
1 dt[y == "b",]
```

```
      x      y      z
   <int> <char> <lgcl>
1:     2      b  FALSE
```

`tidyverse` packages

- `tidyverse` package ecosystem - rich collection of data science packages.
- Designed with consistent interfaces and generally higher usability than base R function.
- Notable packages:
  - `readr` - data input/output (also `readxl` for spreadsheets, `haven` for SPSS/Stata)
  - `dplyr` - data manipulation (also `tidyr` for pivoting)
  - `ggplot2` - data visualisation
  - `lubridate` - working with dates and time
  - `tibble` - enhanced data frame

Tibble - User-friendly Data Frame

- Tibbles are designed to be backward compatible with base R data frames.
- Console printing of tibbles is cleaner (prettified, only first 10 rows by default).
- Tibbles can have columns that themselves contain lists as elements.
- Seamless integration with the `tidyverse` ecosystem.

```
library("tibble")
```

```
tb <- tibble::tibble(
    x = 1:4,
    y = c("a", "b", "c", "d"),
    z = c(TRUE, FALSE, FALSE, TRUE)
)
tb
```

```
# A tibble: 4 × 3
      x y     z
  <int> <chr> <lgl>
1     1 a     TRUE
2     2 b     FALSE
3     3 c     FALSE
4     4 d     TRUE
```

Tibble: Example

- Tibbles work (mostly) like data frames

```
str(tb)
```

```
tibble [4 × 3] (S3: tbl_df/tbl/data.frame)
 $ x: int [1:4] 1 2 3 4
 $ y: chr [1:4] "a" "b" "c" "d"
 $ z: logi [1:4] TRUE FALSE FALSE TRUE
```

```
dim(tb)
```

```
[1] 4 3
```

```
tb[c("x", "z")]
```

```
# A tibble: 4 × 2
      x z
  <int> <lgl>
1     1 TRUE
2     2 FALSE
3     3 FALSE
4     4 TRUE
```

```
tb[tb$y == "b",]
```

```
# A tibble: 1 × 3
      x y     z
  <int> <chr> <lgl>
1     2 b     FALSE
```

Data Manipulation With `dplyr`

- `dplyr` - is one of the core packages for data manipulation in `tidyverse`
- Its principal functions are:
  - `filter()` - subset rows from data
  - `mutate()` - add new/modify existing variables
  - `rename()` - rename existing variable
  - `select()` - subset columns from data
  - `arrange()` - order data by some variable
- For data summary:
  - `group_by()` - aggregate data by some variable
  - `summarise()` - create a summary of aggregated variables

```
1 library("dplyr")
```

dplyr: Subsetting

```
dplyr::filter(tb, y == 'b', z == FALSE)
```

```
# A tibble: 1 × 3
      x y     z
  <int> <chr> <lgl>
1     2 b     FALSE
```

```
# Note that dplyr functions do not require enquoted variable names
dplyr::select(tb, x, z)
```

```
# A tibble: 4 × 2
      x z
  <int> <lgl>
1     1 TRUE
2     2 FALSE
3     3 FALSE
4     4 TRUE
```

dplyr: Modifying Columns

```r
# Data is not modified in-place, you need to re-assign the results
tb <- dplyr::mutate(tb, rand = rnorm(4))
tb
```

```
# A tibble: 4 × 4
      x y     z       rand
  <int> <chr> <lgl>  <dbl>
1     1 a     TRUE  -1.54
2     2 b     FALSE  0.879
3     3 c     FALSE  0.635
4     4 d     TRUE  -1.91
```

```r
tb <- dplyr::rename(tb, random = rand)
```

```r
# We can also use helpful tidyselect functions for more complex rules
dplyr::select(tb, tidyselect::starts_with('r'))
```

```
# A tibble: 4 × 1
  random
   <dbl>
1 -1.54
2  0.879
3  0.635
4 -1.91
```

%>% Operator

- Users of `tidyverse` packages are encouraged to use pipe operator `%>%`.
- It allows to chain data transformations without creating intermediate variables.
- It passes the result of the previous operation as a first first argument to the next.
- Base R now also includes its own pipe operator `|>` but it is still not used as widely.

```
<result> <- <input> %>%
  <function_name>(., arg_1, arg_2, ..., arg_n)

<result> <- <input> %>%
  <function_name>(arg_1, arg_2, ..., arg_n)
```

%>% Operator: Example

```
1 tb
```

```
# A tibble: 4 × 4
      x y     z      random
  <int> <chr> <lgl>  <dbl>
1     1 a     TRUE  -1.54
2     2 b     FALSE  0.879
3     3 c     FALSE  0.635
4     4 d     TRUE  -1.91
```

```
1 tb <- tb %>%
2   dplyr::mutate(random2 = rnorm(4)) %>%
3   dplyr::filter(z == FALSE)
```

```
1 tb
```

```
# A tibble: 2 × 5
      x y     z      random random2
  <int> <chr> <lgl>  <dbl>   <dbl>
1     2 b     FALSE  0.879  0.0139
2     3 c     FALSE  0.635  0.137
```

%>% vs |> Operator

- Since R version 4.1.0, there is a built-in |> (pipe) operator.

```
# Pipe %>% can also be used with non-dplyr functions
# Analogous to tb$x %>% `[`(2)
tb$x %>% .[2]
```

```
[1] 3
```

```
# Base R pipe operator |> is more restrictive
# E.g. tb$x |> `[`(2) doesn't work
# As well as other functions that act as infix operators
tb |> nrow()
```

```
[1] 2
```

Pivoting Data

- Sometimes you want to pivot you data by:
  - Spreading some variable across columns (`tidyr::pivot_wider()`)
  - Gathering some columns in a variable pair (`tidyr::pivot_longer()`)

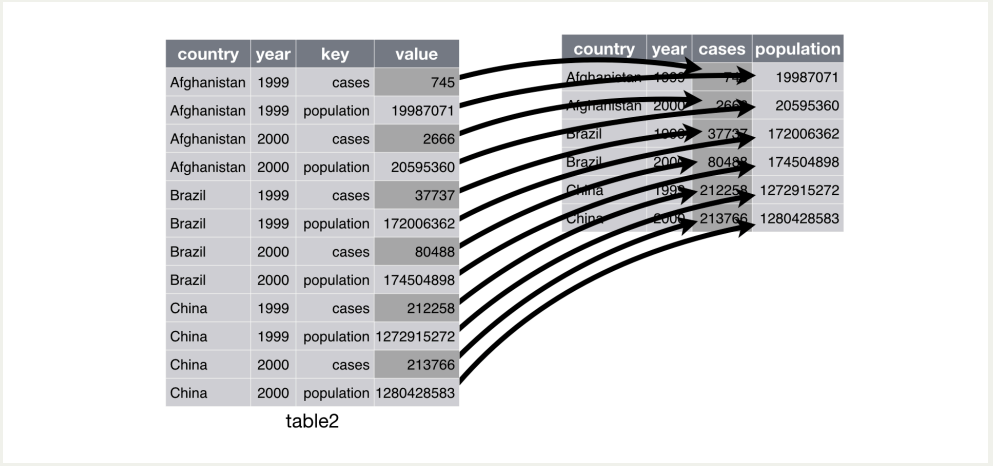


table2

pivot_wider()



table4

pivot_longer()

Pivoting Data: Example

```r
tb2 <- tibble::tibble(
  country = c("Afghanistan", "Brazil"),
  `1999` = c(745, 2666),
  `2000` = c(37737, 80488)
)
tb2
```

```
# A tibble: 2 × 3
  country     `1999` `2000`
  <chr>        <dbl>  <dbl>
1 Afghanistan    745  37737
2 Brazil        2666  80488
```

```r
tb2 <- tb2 %>%
  # Note that pivoting functions come 'tidyr' package
  tidyr::pivot_longer(
      cols = c("1999", "2000"),
      names_to = "year",
      values_to = "cases"
  )
tb2
```

```
# A tibble: 4 × 3
  country     year  cases
  <chr>       <chr> <dbl>
1 Afghanistan 1999    745
2 Afghanistan 2000  37737
3 Brazil      1999   2666
4 Brazil      2000  80488
```

```r
tb2 <- tb2 %>%
  tidyr::pivot_wider(names_from = "year", values_from = "cases")
```

```
3 tb2
```

```
# A tibble: 2 × 3
  country     `1999` `2000`
  <chr>        <dbl>  <dbl>
1 Afghanistan    745  37737
2 Brazil        2666  80488
```

Next

- Tutorial: Data wrangling in R and Python
- Next week: Classes and Object-oriented programming