Data Wrangling

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Data in the real world typically does not arrive ready for analysis. Instead usually manipulate the data in various ways to derive a nice, clean, tidy data frame of rows by columns with all the data values in a column of the same type, such as character strings, integers, or floating point numbers (i.e., with decimal digits).

> *Data Wrangling*: The process of cleaning, tidying, and otherwise preparing data for analysis.

The following examples demonstrate some useful data manipulations that are applicable to all data analysis: subsetting a data table, converting variable types, and variable transformations. Merging data frames, another common data manipulation, is shown elsewhere.

This task of data wrangling is where most data scientists spend most of their time, up to 80% is a common understanding. Larger organizations have added a new job category called data engineer, a specialist in data wrangling and other aspects of handling data, such as data collection.

The material in this and related notebooks covers some of the basic, and most common, data wrangling procedures.

Preliminaries

Packages

```
from datetime import datetime as dt
dt.now().year
print ('Analysis on', now.strftime('%Y-%m-%d'), 'at', now.strftime('%H:%M'))
```

```
Analysis on 2021-06-27 at 14:06
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Read

```
#i = pd.read_excel('data/employee.xlsx')
d = pd.read_excel('http://readstats.com/data/employee.xlsx')
```

```
d.shape
```

```
(37, 9)
```

```
d.head()
```

	Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
0	Richie, Darnell	7.0	M	ADMN	53788.25	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62

Data Frame Row Identifiers

Columns in a data frame represent the variables in the analysis. Columns can be identified be the corresponding variable name or by their numerical position in the data frame.

> *Index*: An integer that specifies the numerical row or column position.

Unfortunately (in my opinion), Python begins all counting with 0 instead of 1. However, the principle of identifying a row or a column by the corresponding integer remains.

In the *d* data frame that contains the read data, as shown above, the default row identifiers are the row indices. However, if there is column of unique identifiers in the data table, that column can be designated as the row identifiers.

In the above *d* data table, the **Name** column appears as any other variable in the data frame. However, **Name** is not a variable per se to analyze but, an ID field, with a unique value for each row. Replace the default integer row labels with the values of the column **Name** with the `set_index()` function.

Note that data manipulation methods typically do not change the original data frame. To save changes, explicitly save the manipulation into a variable, such as below, where the change to the *d* data frame is saved back into the *d* data frame. If there is no assigned variable for the output, the output is directed to the console. You will be able to view the output, but no changes are saved.

```
d = d.set_index('Name')
d.head()
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Richie, Darnell	7.0	M	ADMN	53788.25	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62

Or, set the row names as the data values directly when read with a function such as `read_excel()`. Use the parameter `index_col` to set the column index. The variable **Name** in the original data frame is in the first column, that is, Column 0.

```
d = pd.read_excel('http://readstats.com/data/employee.xlsx', index_col=0)
#d = pd.read_excel('data/employee.xlsx', index_col=0)
d.head()
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Richie, Darnell	7.0	M	ADMN	53788.25	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62

Subset Rows and Columns

A data frame is the Python representation of a rectangular data table with rows and columns. In many situations, we wish to view or analyze just a portion of the entire data frame, a subset, or extraction from the original. Many possibilities of subsetting exist: a single data value (cell), a single row, a single column, and a range of rows or columns.

Express all references to data values in a data frame in terms of those rows and columns. Reference data values within data frame *d* by its rows and columns:

```
d[row_reference, column_reference]
```

The reference can be a corresponding name of a row or column or its associated integer index. If referencing location by name, use the `.loc()` function for "location". Reference the integer index with the `.iloc()` function for "integer location" or "index location". The "weird" part, as with all Python counting, is that counting rows or columns starts with 0 instead of starting at 1.

Note: There is no assignment of the sub-setted information to the new data frame in the examples below, just a display of the requested information.

Select a Single Cell

We see from the original data table that Binh Hoang's salary is $111,074.86.

To display the data value stored in a single, specific cell, specify a single row reference and a single column reference. Here reference the location of the data value by the row name and column name, so use the `.loc` method for "location".

Note that the output is not assigned to another data frame, so the output is directed to the space right below where the cell is located.

```
d.loc['Hoang, Binh', 'Salary']
```

```
111074.86
```

Or, reference the location of the data value by its row and column index according to their integer positions in the data frame with `.iloc`. The data for Binh Hoang are in the third row, Row 2 because Python starts counting at 0. The variable *Salary* is in the fourth column, Column 3.

```
d.iloc[2,3]
```

```
111074.86
```

Select Multiple Cells

A colon, `:`, indicates a range of either rows (before the comma) or columns (after the comma).

```
d2 = d.loc['Wu, James':'Jones, Alissa', 'Gender':'Jobsat']
d2.head()
```

Name	Gender	Dept	Salary	Jobsat
Wu, James	M	SALE	94494.58	low
Hoang, Binh	M	SALE	111074.86	low
Jones, Alissa	F	NaN	53772.58	NaN

When subsetting row or column indices in `pandas` with the colon operator, `:`, always reference the counting system that begins with 0. And then reference the row after the last one you wish to select.

For example, to select the second through the fourth row, the `pandas` row indices are 1 through 3. So specify a row range of 1:4. To select the second through fourth row, specify a column range of 1:5. (Why do the Python people have to complicate such a simple issue as counting?) To illustrate, show here again the first five rows of the *d* data frame.

```
d.head()
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Richie, Darnell	7.0	M	ADMN	53788.25	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62

Now select the second through the fourth rows, getting three rows indicated by indices 1 through 3, and the second through the fifth columns, getting four columns, indicated by indices 1 through 4.

```
d2 = d.iloc[1:4, 1:5]
d2
```

Name	Gender	Dept	Salary	Jobsat
Wu, James	M	SALE	94494.58	low
Hoang, Binh	M	SALE	111074.86	low
Jones, Alissa	F	NaN	53772.58	NaN

Subset Rows

Select a Single Row

The `:` by itself after the comma, that is, no columns specified, indicates to retrieve *all* columns for the specified row(s). The colon for columns is optional, but good practice to include.

```
d.loc['Hoang, Binh', :]
```

```
Years         15
Gender         M
Dept        SALE
Salary    111074
Jobsat       low
Plan           3
Pre           96
Post          97
Name: Hoang, Binh, dtype: object
```

Unfortunately, Python's specification of row and column ranges with the `:` is unnecessarily complex. The straightforward way to proceed is to identify each row or column with its ordinal position, that is, counting from 1. Instead, Python starts counting from 0, so the index of the first column and of the first row is 0.

Another complication is that the range itself indicates the beginning index but does not include the ending index. To illustrate, a range of 2:3, if placed before the comma, references only Row index 2, which is the third row.

```
d.iloc[2:3, :]
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97

Specify a row without a range results in the vertical listing of the information for that row, here Row 3.

```
d.iloc[2, :]
```

```
Years         15
Gender         M
Dept        SALE
Salary    111074
Jobsat       low
Plan           3
Pre           96
Post          97
Name: Hoang, Binh, dtype: object
```

Subset Rows by Data Values

Retrieve just those rows of data that match a logical condition. Here identify all rows of data for employees with a Salary more than $100,000 per year. The first example only displays the result. The second example creates a new data frame with the result, and then displays the subset data frame.

This first example invokes the most straightforward method of selecting rows that satisfy a logical condition: The `query()` function. Note the logical condition is enclosed in quotes.

```
d.query('Salary > 100000')
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Currell, Trevon	21.0	M	SALE	134419.23	low	1	97	94
James, Yvette	18.0	F	ADMN	122582.38	low	3	70	70

The `query()` function may be more straightforward, but there is another expression for the extraction that many people use. This expression involves repeating the name of the data frame, which becomes awkward for long data frame names and a logical expression that involves multiple variables. Still, this form is frequently encountered in `pandas` data manipulation.

```
dd = d[d['Salary'] > 100000]
dd
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Currell, Trevon	21.0	M	SALE	134419.23	low	1	97	94
James, Yvette	18.0	F	ADMN	122582.38	low	3	70	70

Here query all Males in *Finance*. The ampersand, `&`, indicates "and". Because the entire expression is enclosed in single quotes, `'`, enclose character constants within the string with double quotes, `"`.

```
d.query('Dept == "FINC" & Gender == "M"')
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Shepperd, Cory	14.0	M	FINC	95027.55	low	3	66	73
Link, Thomas	10.0	M	FINC	66312.89	low	1	83	83
Cassinelli, [illegible]	[illegible]	M	FINC	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

The `str.contains()` function selects values that contain a specified character string. However, the method only works with complete data, so first remove missing data values in *d* with the `dropna()` function.

```
d2 = d.dropna()
d2[d2['Dept'].str.contains('F')]
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Downey, Deborah	7.0	F	FINC	57139.90	high	2	90	86
Shepperd, Cory	14.0	M	FINC	95027.55	low	3	66	73
Link, Thomas	10.0	M	FINC	66312.89	low	1	83	83

Subset Columns

The easiest way to subset variables, select columns, is with the `filter()` function.

```
d.head()
```

Name	Years	Gender	Dept	Salary	Jobsat	Plan	Pre	Post
Richie, Darnell	7.0	M	ADMN	53788.25	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62

In many situations in data analysis, we wish to refer simultaneously to multiple values instead of a single value. With Python in general and `pandas` specifically, define a vector of values using square brackets `[ ]`.

> *Vector*: A variable that consists of multiple values.

Note to R users: R uses the `c()` function to accomplish the same result.

In this example, create a vector of two variable names. To emphasize the definition of a vector, define the vector separately from the call to the `filter()` function.

```
d2 = d.filter(['Gender', 'Salary'])
d2.head()
```

Name	Gender	Salary
Richie, Darnell	M	53788.25
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
[illegible]	[illegible]	[illegible]

To emphasize the definition of a vector, define the vector of variable names separately from the call to the `filter()` function.

```
my_vector = ['Gender', 'Salary']
d2 = d.filter(my_vector)
d2.head()
```

Name	Gender	Salary
Richie, Darnell	M	53788.25
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
[illegible]	[illegible]	[illegible]

To select variables by their names can also use the more general `loc()` function. Need to specify both rows and columns with `loc`, separated by a colon `:`. If not simultaneously selecting rows, put nothing before the `:`.

```
d2 = d.loc[:, ['Gender', 'Salary']]
d2.head()
```

Name	Gender	Salary
Richie, Darnell	M	53788.25
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
[illegible]	[illegible]	[illegible]

Or, can use `iloc` for "index location" if specifying the relevant numerical index.

```
d2 = d.iloc[:, [1,3]]
d2.head(1)
```

	Gender	Salary
Name		
Ritchie, Darnell	M	53788.26
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
Downs, Deborah	F	57139.90

Select by Variable Type

The select function for selecting by variable types is `select_dtypes()`, either the parameter `exclude` or `include`.

```
d_num = d.select_dtypes(exclude='object')
d_num.head()
```

	Years	Salary	Plan	Pre	Post
Name					
Ritchie, Darnell	7.0	53788.26	1	82	92
Wu, James	NaN	94494.58	1	62	74
Hoang, Binh	15.0	111074.86	3	96	97
Jones, Alissa	5.0	53772.58	1	65	62
Downs, Deborah	7.0	57139.90	2	90	86

```
d_obj = d.select_dtypes(include='object')
d_obj.head()
```

	Gender	Dept	Joblvl
Name			
Ritchie, Darnell	M	ADMN	med
Wu, James	M	SALE	low
Hoang, Binh	M	SALE	low
Jones, Alissa	F	NaN	NaN
Downs, Deborah	F	FINC	high

Chained Functions

Likely the most straightforward to accomplish data frame subsets is with the `query()` and `filter()` functions. However, when doing multiple function calls, one after the other, you can chain these calls into a single call. This chaining elucidates a complex, multi-step data manipulation process with highly readable, structured code.

To specify a set of chained functions, include the entire expression within parentheses (), then separate each function call on its own line. In this example, subset by rows, then by column, sort the Salary column with the `sort_values()` function.

```
(d
 .query('Salary > 100000')
 .filter(['Gender', 'Salary'])
 .sort_values('Salary', ascending=False)
)
```

	Gender	Salary
Name		
Connell, Trevon	M	134419.23
James, Leslie	F	120563.38
Hoang, Binh	M	111074.86
Capelle, Adam	M	108138.45

Delete Rows or Columns

Delete a Row

Start with 37 rows of data.

```
d.shape
```

```
(37, 8)
```

Use the `drop()` function to delete a row by row name, to result in 36 rows.

```
d2 = d.drop('Wu, James')
d2.shape
```

```
(36, 8)
```

Delete a row by row index, here the second row, to result in 36 rows.

```
d2 = d.drop(d.index[1])

d2.shape
```

```
(36, 8)
```

The data for James Wu is gone.

```
d2.head()
```

	Years	Gender	Dept	Salary	Joblvl	Plan	Pre	Post
Name								
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86
Akhart, Anbar	6.0	F	ADMN	69441.93	high	2	100	100

Delete a Column

The function `drop()` deletes a row or column from the data frame, as specified by the `axis` parameter. The default value of `axis` is 'rows', so if dropping a column, need to explicitly specify `axis='columns'` to drop the variable from the resulting data frame of only numeric variables because it is an integer coded categorical variable. Dropping that variable leaves only continuous variables.

```
d_num = d2.drop('Plan', axis='columns')
d_num.head()
```

	Years	Gender	Dept	Salary	Joblvl	Pre	Post
Name							
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	82	92
Hoang, Binh	15.0	M	SALE	111074.86	low	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	90	86
Akhart, Anbar	6.0	F	ADMN	69441.93	high	100	100

Other Issues

A kind of "weird" issue exists (especially if you are used to R). Subsets (slices) of data frames by default do not create clean copies. The new data frame is still linked to the original data frame. Changes to the newly created data frame change the original as well! Modifications to the data or indices of the copy are reflected back to the original object.

To make the subset data frame completely independent of d, what pandas calls a deep copy, invoke the subset extraction with the `copy()` function. This function is not needed if there will not be further manipulation of the contents of the newly created data frame. Unless memory is an issue for very large data sets, or you know that there will be no further modification, using copy() is a good practice.

```
d2 = d.iloc[:, 'Salary'].copy()
d2.head()
```

```
Name
Ritchie, Darnell    53788.26
Wu, James           94494.58
Hoang, Binh        111074.86
Jones, Alissa       53772.58
Downs, Deborah      57139.90
Name: Salary, dtype: float64
```

A short-hand specification to select a column does not call any method or function and only includes the names of the relevant columns. When learning a language, better to focus on the more complete implementation of a concept. However, there is a need to know this abbreviated form because it appears often in real-world applications.

```
d2 = d['Salary']
d2.head()
```

```
Name
Ritchie, Darnell    53788.26
Wu, James           94494.58
Hoang, Binh        111074.86
Jones, Alissa       53772.58
Downs, Deborah      57139.90
Name: Salary, dtype: float64
```

Variable Transformation

Transform the values of a numerical variable with an equation, a function that specifies how each value is to be transformed. Transform the values of a categorical variable with a recoding that specifies how each individual value is to be replaced with a new value.

Numeric Variable

To transform the values of a numeric variable is straightforward. Enter the corresponding equation that defines the transformation. Refer to a variable within a data frame with the data frame name, such as d, and then the variable name within quotes and square brackets. For example, to refer to the Salary variable in the d data frame: `d['Salary']`.

This example creates a new variable, SalaryK000, defined as the original Salary variable with values divided by 1000, rounded to two decimal digits. The new variable is added to the already existing variables in the d data frame. Running the equation creates the values of the new variable for all rows of data in the data frame.

```
d['SalaryK00'] = round(d['Salary'] / 1000, 2)
d.head(1)
```

	Years	Gender	Dept	Salary	Joblvl	Plan	Pre	Post	SalaryK00
Name									
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	M	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97	111.07
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62	53.77
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86	57.14

Categorical Variable

The function `replace()` recodes individual values of a categorical variable. Parameter `to_replace` indicates the values to be replaced, and parameter `value` indicates the replacement value. Here replace across the entire data frame. Here recode both values of Gender with one statement.

```
d.dtypes
```

```
Years         float64
Gender         object
Dept           object
Salary        float64
Joblvl         object
Plan            int64
Pre             int64
Post            int64
SalaryK00     float64
dtype: object
```

This way to recode with `replace()` replaces any 'F' and 'M' in the entire data table, which turns out to be just for the values of Gender in this example.

```
d_obj = d.replace(to_replace='F', value='Female', inplace=True)
d_obj.head(1)
```

	Years	Gender	Dept	Salary	Joblvl	Plan	Pre	Post	SalaryK00
Name									
Ritchie, Darnell	7.0	Male	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	Male	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	Male	SALE	111074.86	low	3	96	97	111.07
Jones, Alissa	5.0	Female	NaN	53772.58	NaN	1	65	62	53.77
Downs, Deborah	7.0	Female	FINC	57139.90	high	2	90	86	57.14

The following use of `replace()` targets just values of the variable Gender. The curly brackets { and } indicate a specific `pandas` data type called a dictionary. As with any dictionary there is a keyword, and then the meaning. For example, the keyword 'F' has meaning 'Female'.

This example presents a dictionary within a dictionary. The keyword Gender as meaning 'F' and 'M', which each has their own meaning.

```
dd = d.replace({'Gender': {'F': 'Female', 'M': 'Male'}})
dd.head()
```

	Years	Gender	Dept	Salary	Joblvl	Plan	Pre	Post	SalaryK00
Name									
Ritchie, Darnell	7.0	Male	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	Male	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	Male	SALE	111074.86	low	3	96	97	111.07
Jones, Alissa	5.0	Female	NaN	53772.58	NaN	1	65	62	53.77
Downs, Deborah	7.0	Female	FINC	57139.90	high	2	90	86	57.14

Rename a Variable

Use the `rename()` function, which allows to specify both the old name and the new name. If defining a new data frame, safer to add the `copy()` function to make sure it is a deep copy with no relation to the original data frame. Or, if wishing to just change the names in the current data frame, no need to set it to another data frame. Set the `inplace` parameter to `True` instead.

In this example, a Python dictionary defines the values to replace each specified individual value. That is, replace the variable name Dept with Section, and Joblvl with Satisfaction.

```
d2 = d.rename(columns = {'Dept': 'Section',
                         'Joblvl': 'Satisfaction'})
d2.head()
```

	Years	Gender	Section	Salary	Satisfaction	Plan	Pre	Post	SalaryK00
Name									
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	M	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97	111.07
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62	53.77
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86	57.14

Convert to Category Variable Type

In any data analysis, always be aware of two fundamental types of variables: Numerical and categorical.

> *Categorical variable*: A variable that consists of only a relatively small number of unique values.

An example of a categorical variable is the department in which an employee works, such as sales, marketing, etc. Another example is eye color, with values of Blue, Green, Brown, Black, and Gray. The values of categorical variables may also be integers, such as the responses to an attitude question on a survey from Strongly Disagree to Strongly Agree encoded as as integers from 1 to 7.

To represent the values of a categorical variable, Python presents the `category` variable type. Variables with character string values are initially read as type `object`. Variables with integer values are initially read as type `int64`. If the variable is categorical, then best to convert the variable to type `category`. This conversion saves memory, and can change the order of the display of the categories (values).

Assign a new variable type with the `astype()` function.

```
d = pd.read_excel('http://jonathantx.com/data/employee.xlsx', index_col=0)
#d = pd.read_excel('data/employee.xlsx')

d_Gender = d.Gender.astype('category')
d_Dept = d.Dept.astype('category')
d_Joblvl = d.Joblvl.astype('category')
d_Plan = d.Plan.astype('category')
d.dtypes
```

```
Years       float64
Gender       object
Dept         object
Salary      float64
Joblvl       object
Plan          int64
Pre           int64
Post          int64
dtype: object
```

> *Loop*: A computer instruction that repeats until a specified condition is reached.

Python has a type of loop called a `for` loop, which processes one or more expressions over a range of values.

Here use a `for` loop to consider each variable one at a time instead of a separate equation for each variable, as was done in the previous cell. The loop applies the `astype()` function to each of the specified variables, beginning with Gender. The expression `dict()` refers to each specified variable, one at a time, beginning with Gender and ending with Plan.

```
d = pd.read_excel('http://jonathantx.com/data/employee.xlsx')
#d = pd.read_excel('data/employee.xlsx')
for i in ['Gender', 'Dept', 'Joblvl', 'Plan']:
    d[dict] = d[dict].astype('category')
d.dtypes
```

```
Name        object
Years      float64
Gender    category
Dept      category
Salary     float64
Joblvl    category
Plan      category
Pre          int64
Post         int64
dtype: object
```

```
d.head()
```

	Name	Years	Gender	Dept	Salary	Joblvl	Plan	Pre	Post
0	Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
4	Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Binning

Pandas function `qcut()` bins a continuous variable into discrete categories according to the specified quantiles. The optional `labels` parameter provides names for the bins, otherwise integers numbered from 0.

```
d2['Sal_bin'] = pd.qcut(d2['Salary'], q=[0,.25,.75,.1], 
                        labels=['Low', 'Med', 'High', 'Top'])
d2.head(5)
```

	Years	Gender	Section	Salary	Satisfaction	Plan	Pre	Post	SalaryK00	Sal_bin
Name										
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92	53.79	Low
Wu, James	NaN	M	SALE	94494.58	low	1	62	74	94.49	Top
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97	111.07	Top
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62	53.77	Low
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86	57.14	Med
Akhart, Anbar	6.0	F	ADMN	69441.93	high	2	100	100	69.44	Med