|  |
| --- |
| E-mail: easydatabase at gmail dot com |
| **Python Pandas DataFrame Cookbook** |
| for Pandas module |
|  |
|  |
|  |

|  |
| --- |
|  |

*Yingyao Zhou  
  
Document Version 1.02   
Updated on Mar 22, 2016*

*Updated on Nov 26, 2022, for Python 3*

*Copyright © 2013 Yingyao Zhou. All rights reserved.*

Table of Contents

[1 Introduction 4](#_Toc120372301)

[2 Input 7](#_Toc120372302)

[2.1 Reading from a CSV file 7](#_Toc120372303)

[2.2 Reading from a TSV file 7](#_Toc120372304)

[2.3 Reading from a database 7](#_Toc120372305)

[2.4 Converting a memory string into a table object 8](#_Toc120372306)

[2.5 Reading from a compressed data file 8](#_Toc120372307)

[2.6 Writing to a compressed data file 8](#_Toc120372308)

[2.7 Reading/Writing an Excel file 8](#_Toc120372309)

[3 Output 8](#_Toc120372310)

[3.1 Writing a CSV file 8](#_Toc120372311)

[3.2 Writing a TSV file 9](#_Toc120372312)

[3.3 Displaying a table in HTML 9](#_Toc120372313)

[4 Accessing Table 11](#_Toc120372314)

[4.1 Getting a table cell 11](#_Toc120372315)

[4.2 Getting table dimensions 12](#_Toc120372316)

[4.3 Looping through table rows 12](#_Toc120372317)

[4.4 Getting a table row 12](#_Toc120372318)

[4.5 Getting a table column 12](#_Toc120372319)

[4.6 Getting all column names 12](#_Toc120372320)

[4.7 Modifying a table cell 12](#_Toc120372321)

[4.8 Adding a new row 13](#_Toc120372322)

[4.9 Deleting a row 13](#_Toc120372323)

[4.10 Deleting rows 13](#_Toc120372324)

[4.11 Adding a new column 13](#_Toc120372325)

[4.12 Deleting a column 13](#_Toc120372326)

[4.13 Moving/Reordering columns 13](#_Toc120372327)

[4.14 Replacing a column 14](#_Toc120372328)

[4.15 Renaming a column 14](#_Toc120372329)

[5 Initializing Table 14](#_Toc120372330)

[5.1 Initializing an empty table 14](#_Toc120372331)

[5.2 Initializing a table by rows 14](#_Toc120372332)

[5.3 Initializing a table by columns 14](#_Toc120372333)

[6 Table Filtering and Sorting 14](#_Toc120372334)

[6.1 Filtering rows 14](#_Toc120372335)

[6.2 Filtering rows by a Boolean mask array 15](#_Toc120372336)

[6.3 Getting a subset of a table 15](#_Toc120372337)

[6.4 Sorting a table by multiple columns 15](#_Toc120372338)

[7 Manipulating Two Tables 15](#_Toc120372339)

[7.1 Joining two tables 15](#_Toc120372340)

[7.2 Merging two tables row-wise 15](#_Toc120372341)

[7.3 Merging two tables column-wise 15](#_Toc120372342)

[8 Transforming a Table 16](#_Toc120372343)

[8.1 Reshaping – melting and casting for table statistics 16](#_Toc120372344)

[8.2 Grouping a table with aggregation functions 19](#_Toc120372345)

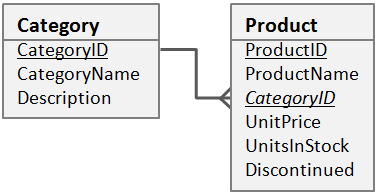
[9 Read Excel File 20](#_Toc120372346)

[9.1 Parse my\_product.xlsx 20](#_Toc120372347)

# Introduction

The DataFrame structure is a wrapper on top of a 2-d numpy array. Each row and each column are indexed. Row index typically are the record keys (same as database), if no key is available, row index is default to integer counting.

In this cookbook, we often use two tables obtained from the Microsoft Northwind sample dataset: “Category” and “Product”.



Category consists of eight food categories, with CategoryID as its primary key. Product consists of 77 products, where ProductID is its primary key and CategoryID is a foreign key pointing to the Category table (see diagram above). Each category can have multiple products. The first five rows of the two tables are shown below in the comma-separated value (CSV) format:

CategoryID,CategoryName,Description  
1,Beverages,"Soft drinks, coffees, teas, beers, and ales"  
2,Condiments,"Sweet and savory sauces, relishes, spreads, and seasonings"  
3,Confections,"Desserts, candies, and sweet breads"  
4,Dairy Products,Cheeses  
5,Grains/Cereals,"Breads, crackers, pasta, and cereal"

ProductID,ProductName,CategoryID,UnitPrice,UnitsInStock,Discontinued  
1,Chai,1,18,39,FALSE  
2,Chang,1,19,17,FALSE  
3,Aniseed Syrup,2,10,13,FALSE  
4,Chef Anton's Cajun Seasoning,2,22,53,FALSE  
5,Chef Anton's Gumbo Mix,2,21.35,0,TRUE

First, let us get a taste of what DataFrame can do.

**Problem**

Assume we are interested in knowing the top three categories that have the largest total value of all in-stock active products.

**Solution**

**import pandas as pd**

**import util**

**# read all products into $t\_product Data::Table object**

**t\_product = pd.read\_csv("Product.csv")**

**# find all in-stock products**

**t\_product = t\_product[ ~t\_product['Discontinued']]**

**# calculate total cost of products in each category**

**tot=t\_product.groupby('CategoryID').apply(lambda x: (x['UnitsInStock']\*x['UnitPrice']).sum())**

**t\_category\_cost=pd.DataFrame({'CategoryID':tot.index, 'TotalCost':tot.values})**

**# read all categories into $t\_category**

**t\_category = pd.read\_csv("Category.csv");**

**# obtain a table of columns: CategoryID,CategoryName,Description,TotalCost**

**t\_category[:3].display()**

**t\_category\_cost[:4].display()**

**t = pd.merge(t\_category, t\_category\_cost, left\_on='CategoryID', right\_on='CategoryID')**

**t=t.sort\_values(by='TotalCost', ascending=False)**

**t.reindex(range(len(t)))**

**t=t.iloc[:3]**

**#print(t.to\_string(index=False, float\_format=(lambda x: '%.2f' % x)))**

**from io import StringIO**

**output = StringIO()**

**t.to\_csv(output, index=False, sep="\t")**

**print(output.getvalue())**

**# outputs  
*CategoryID CategoryName Description TotalCost  
8 Seafood Seaweed and fish 13010.35  
1 Beverages Soft drinks, coffees, teas, beers, and ales 12390.25  
2 Condiments Sweet and savory sauces, relishes, spreads 12023.55***

**Discussion**

We need to read in all product data, filter out discontinued ones, calculate the TotalCost for each product category, append additional category annotation data (CategoryName and Description), keep the top three, and report.

From\_csv() reads in a CSV/TSV file, it automatically detects the file format, including the presence of a column header. We then only keep the records where the Discontinued field equals **"**FALSE**"** using conditional expression. A new table t\_category\_cost is constructed by calculating TotalCost for each product category using groupby(). We merge in additional category annotation data using merge(), then sort\_index() the resultant table and keep the top three categories. The final data is reported in tab-delimited value (TSV) format.

**Conventions**

Python code is printed with Consolas font. Screen output is printed with *Consolas Italic*.

When we run scripts in this book, always remember to include:

import numpy as np  
import pandas as pd  
import util

To save space, we always omit these lines afterwards.

As we use two example tables, Category and Product, so often, we may also directly use t\_product and t\_category without initialization:

**t\_product = pd.read\_csv("Product.csv");  
t\_category = pd.read\_csv("Category.csv");**

Variable t is often used to represent a generic DataFrame object.   
  
The files needed for this tutorial can be downloaded from: https://github.com/data2code/pandas

To use git: git clone https://github.com/data2code/pandas.git

# Input

## Reading from a CSV file

**Solution**

**t\_product = pd.read\_csv("Product.csv");**

**Discussion**

The first parameter is the file name. header=0 by default, which assumes the first line of Product.csv is for column headers. If the file does not contain a header line (header=None), the columns will be automatically named as 0, 1, 2, etc. We may also supply our own column name:

**t\_category = pd.read\_csv("Category.csv", skiprows=1, names=["ID", "Name", "Comment"])**

In the above example, the header line in the file will be ignored and the supplied column names are used instead.

## Reading from a TSV file

**Solution**

**# Pretend Product.tsv is a tab-delimited file  
t\_product = pd.read\_table("Product.tsv") # more consistent with read\_csv  
t\_product = pd.read\_csv("Product.tsv", sep="\t")**

## Reading from a database

**Solution**

**import db # use a GNF in-house module   
mydb=db.DB('IMCPROD')  
t = mydb.from\_sql(con, "select \* from Product where ProductID > ?, params=[ 12 ])**

**Discussion**

Often we need to fetch records from a database; my.from\_sql() takes a database handler as the first parameter, and a SQL statement as the second. The SQL statement does not have to be a SELECT statement, if a statement such as INSERT/UPDATE/DELETE, the return DataFrame will simply be None.

If the SQL expects parameter binding, from\_sql() takes an array reference as the third parameter.

**t = mydb.from\_sql(  
 "SELECT \* FROM Product WHERE ProductName like ? AND Discontinued=?",  
 params=['BOSTON%', 'FALSE']  
)**

In MySQL, a primary key of a record may often be set to AUTO\_INCREMENT, i.e., database will assign the key for us. In this case, right after we INSERT a new record, we may use LAST\_INSERT\_ID() to obtain the value of the newly assigned key.

**t = mydb.from\_sql("SELECT LAST\_INSERT\_ID()");  
print(t.iloc[0,0])**

*I need to check this part later: Notice fromSQL() reads in the whole table from database all at once, which makes it really convenient. A hint, if we reads large amount of data across a slow network, read data in bulk might boost performance, try to set $dbh->{RowCacheSize}=16\*1024 (this is really a topic belongs to the DBI module).*

## Converting a memory string into a table object

**Solution**

**from io import StringIO**

**output = StringIO('A,B\n1,2')**

**pd.read\_csv(output)  
# outputs**

**A B**

1. **1 2**

## Reading from a compressed data file

**Solution**

**t\_product = pd.read\_csv('Product.csv.gz')**

## Writing to a compressed data file

**Solution**

**import util # my own util.py**

**t\_product = pd.to\_csv('Product.csv.gz', index=False))**

## Reading/Writing an Excel file

**Solution**

#Need to read Pandas doc for more details, I did not try these

xls = pd.ExcelFile('Product.xlsx')  
t\_product = xls.parse('Product', index\_col=None, na\_values=['NA'])  
t\_product.to\_excel('product\_copy.xlsx', sheet\_name='Product', index=False)

# Output

## Writing a CSV file

**Solution**

**t\_product.to\_csv("Product.csv", index=False);**

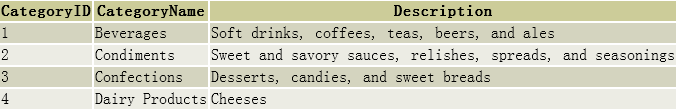
## Writing a TSV file

**Solution**

**t\_product.to\_csv("Product.csv", index=False, sep="\t");**

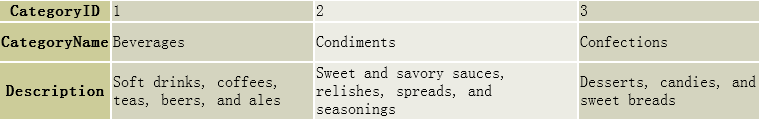
## Displaying a table in HTML

**Solution**

**import util  
t\_category.html()  
**

**Discussion**

**If we want to show the table in wide mode, i.e., each row is displayed as a column (typically for a fat table of many columns but few rows), use**

**import util  
t\_category.html(portrait=True)  
**

**By default, the table displayed uses a default color theme, where odd and even rows have alternative-color background and header has a darker-color background. We can fine tune the HTML table by providing additional parameters.**

**The first parameter can be a color array reference, e.g., to generate the colors in the snapshot, use ["#D4D4BF","#ECECE4","#CCCC99"] to specify colors for odd rows, even rows, and column header, respectively (default colors are shown in the screenshots above). As CSS now becomes so popular, one would probably choose to specify a hash reference that defines CSS classes for odd, even, and header tags, by default the class names are:**

**{"even" : "data\_table\_even", "odd" => "data\_table\_odd", "header" => "data\_table\_header"}**

**The method takes additional parameters defining properties for tags including <TABLE>, <TR>, <TH>, and <TD> in the form of hash reference.**

**t\_category.html(  
 {"even" : "my\_even", "odd" : "my\_odd", "header" : "my\_header"},  
 # properties for <TABLE> tag  
 {"border" : '1'},  
 # properties for <TR> tag  
 {"align" : 'left'},   
 # properties for <TH> tag   
 {"align" : 'center'},  
 # properties for <TD> tag  
 {  
 "CategoryName" : 'align="right" valign="bottom"',  
 2 => 'align="left"'  
 });  
  
# outputs HTML code  
*<table border="1">  
<thead>  
<tr align="left" class="my\_header"><th align="center">CategoryID</th><th align="center">CategoryName</th><th align="center">Description</th></tr>  
</thead>  
<tbody>  
<tr align="left" class="my\_odd"><td>1</td><td align="right" valign="bottom">Beverages</td><td align="left">Soft drinks, coffees, teas, beers, and ales</td></tr>  
<tr align="left" class="my\_even"><td>2</td><td align="right" valign="bottom">Condiments</td><td align="left">Sweet and savory sauces, relishes, spreads, and seasonings</td></tr>  
...  
</tbody>  
</table>***

**In this example, it adds CSS classes: my\_header, my\_oldd, my\_even to table header row, odd rows, and even rows, respectively. The actually colors for these classes are typically defined in .css files included somewhere else in the HTML page. It generates <table border="1">, because we provide {border => "1"} as the parameter and one can certainly specify more name-value pairs in this hash structure to further control <table>. <tr algin="left">, <th align="center"> are the results of corresponding parameters. The pattern here is: each hash key becomes the name of the tag attributes and hash value becomes the attribute value. <td> controlling parameter takes the format that the keys can be either column names or column indices (the column index is the numerical position of a column, e.g., the first column has an index of 0, the value is the second column has an index of 1, etc), and the hash values are the tag attributes go into the corresponding <td> tag. In our example above, we add 'align="right" valign="bottom"' to each <td> tag corresponding to the CategoryName column, and add 'align="left"' to column 2 (the Description column, the third column in the table, with a column index of 2).**

**Nowadays, one almost always wants to provide a {class => "classname"} and then define class properties in .css files. So instead of CategoryName => 'align="right" valign="bottom"', it almost makes more sense to write CategoryName => 'class="myCategoryName"', and define myCategoryName within a .css file.**

**We can also control class and style for each individual cell within a table, include header cells. The way to accomplish this is by providing a callback method. The callback method can take the following arguments: my\_callback(tag\_dict, row\_index, col\_index, col\_name, dataframe). The tag\_dict is the current dict of tags for the given cell, row\_index, col\_index and dataframe allows us to determine which cell is the target cell, col\_name is provided for convenience. Notice, if row\_index is -1, it indicates the target cell is a column header cell. The following example which highlight the cells for expensive products in orange and cheap ones in blue, color the cells for discontinued item in gray and active products in purple.**

**t=pd.read\_csv('Product.csv')**

**t=t[:6]**

**def callback(tag, r, c, col, df):**

**if r>-1 and col=='UnitPrice':**

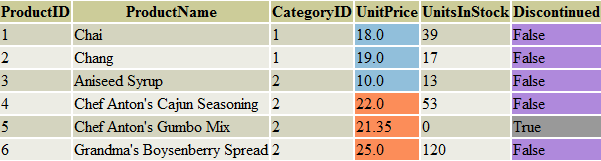
**tag['style']= 'background-color:#fc8d59;' if df.iloc[r,c]>20 else 'background-color:#91bfdb;'**

**if r>-1 and col=='Discontinued':**

**tag['style']= 'background-color:** **#999999;' if df.iloc[r,c] else 'background-color:#af89dc;'**

**return tag**

**print(t.html(["#D4D4BF","#ECECE4","#CCCC99"], callback=callback))**



**Method**

html(), html2()

# Accessing Table

## Getting a table cell

**Solution**

**t\_product.loc[0]['ProductID']**

**t\_product['ProductID'][0]**

**t\_product['ProductID'].iloc[0]**

**t\_product.values[0,0] # use iloc instead  
t\_product.iloc[0, 0]**

**# returns 1**

**Discussion**

**The coordinate of a cell is defined by its row index and column names. Row index are not necessary the row number, column names maybe integer. This is the confusing part.**

**When we initially construct t\_product, since no row index is specified, it automatically use row number as row index, 0, 1, … However, if we sort the t\_product (or delete some rows):**

**t\_product[:3] # shows the top 3 rows, which are ProductID=76, 75, 74**

**t\_product.loc[0] # do not return ProductID 76, but return ProductID 1  
# to get the first row, use  
t\_product.iloc[0]  
t\_product[0:1] # returns a single-row table**

**t\_product.values[0]  
t\_product.loc[t\_product.index[0]]**

**# or reindex it  
t\_product.index=range(len(t\_product))**

**t\_product.loc[0]**

## Getting table dimensions

**Solution**

**len(t\_product)  
# returns 77  
len(t\_product.columns)  
# returns 6**

## Looping through table rows

**Solution**

**To loop through all rows in a table:**

**for i,r in t\_product.iterrows():  
 print(i, r)**

**# reverse iteration  
for i,r in t\_product[::-1].iterrows():  
 print(i,r)**

## Getting a table row

**Solution**

**See 4.1**

## Getting a table column

**Solution**

**product\_names = t\_product["ProductName"];  
t\_product.ProductName  
# t\_product[2] tries to return a column named 2, not the row index 2!  
# to return column index 2, use  
t\_product[t\_product.header()[2]]**

## Getting all column names

**Solution**

**import util  
t\_product.header()**

## Modifying a table cell

**Solution**

**t\_product.loc[0, "ProductName"]="New Product Name for ProductID 1"  
t\_product["ProductName"]="New Name" # change the whole column  
t\_product.loc[:5, "ProductName"]="New Name" # change first 5 rows  
t\_product.loc[:,"ProductName"]="New Name" # change the whole column**

**Performance discussion  
It appears .ix is good at read a table cell, .at is fast at modifying a table cell.  
The fastest way, however, is to use .get\_value(row,col) and .set\_value(row,col,val)**

## Adding a new row

**Solution**

**t=t\_product.append({'ProductID':78, 'ProductName':"Extra Tender Tofu", 'CategoryID':7, 'UnitPrice':23.25, 'UnitInStock':20, 'Discontinued':False}, ignore\_index=True)  
# or use [{…}] if append multiple rows**

**Discussion**

Append is a slow operation, as it changes the underlying numpy array. If we need to append lots of rows one by one, we should instead collection new rows in a list and use pd.concat to merge the list into one DataFrame.

## Deleting a row

**Solution**

**#delete the last row  
t\_product.drop(len(t\_product), axis=0)  
# use t\_product.index[len(t\_product)], if index is not continuous**

## Deleting rows

**Solution**

**T\_product.drop(t\_product.index[:5], axis=0)**

## Adding a new column

**Solution**

**t\_category["Code"]= ["BV","CD","CF","DR","GC","MP","PR","SF"]  
t\_category["Comment"]="No comment yet"**

## Deleting a column

**Solution**

**del t\_product["Discontinued"]   
t\_product.drop(["UnitsInStock","Discontinued"], axis=1)**

## Moving/Reordering columns

**Solution**

**t\_product=t\_product.reindex(columns=t\_product.columns[::-1])  
# or use util  
import util  
t\_product.move\_column('Discontinued', 0)**

**t\_product.move\_before(['CategoryID', 'Discontinued'], 'ProductID')  
t\_product.move\_after(['CategoryID', 'Discontinued'], 'ProductID')**

## Replacing a column

**Solution**

**t\_product['Discontinued']=t\_product['Discontinued'].apply(lambda x: 'Y' if x else 'N')**

## Renaming a column

**Solution**

**t\_product.rename(columns={"ProductName": "Product\_Name"});  
# use rename2 defined in util package, rename2 is much faster  
t\_product.rename2({"ProductName": "Product\_Name"});**

# Initializing Table

## Initializing an empty table

**Solution**

**pd.DataFrame( columns=["A","B","C"], dtype=float)  
# dtype does not seem to matter, it seems to be determined by first append**

## Initializing a table by rows

**Solution**

**>>> pd.DataFrame([(1,'a'),(2,'b'),(3,'c')], columns=["A","B"])**

**A B**

**0 1 a**

**1 2 b**

**2 3 c**

**>>> pd.DataFrame([{'A':1,'B':'a'},{'A':2,'B':'b'},{'A':3,'B':'c'}])**

**A B**

**0 1 a**

**1 2 b**

**2 3 c**

## Initializing a table by columns

**Solution**

**pd.DataFrame({"A":[1,2,3],"B":[1.5,2.5,3.5],"C":["x","y","z"]}]**

# Table Filtering and Sorting

## Filtering rows

**Solution**

**t\_expensive = t\_product[(t\_product['UnitPrice'] >20) & (~t\_product['Discontinued'])]**

## Filtering rows by a Boolean mask array

**Solution**

cheap = t\_product['UnitPrice'].apply(lambda x: x<20)  
t\_product[cheap]

## Getting a subset of a table

**Solution**

**t\_product.ix[3:8, [**'ProductID','UnitPrice'**]]**

## Sorting a table by multiple columns

**Solution**

**t\_product.sort\_values(['Discontinued', 'UnitPrice'], ascending=[False, True], inplace=True)**

# Manipulating Two Tables

## Joining two tables

**Solution**

**t\_product.merge(t\_category, how=**"left"**, left\_on=[**"**CategoryID**"**],right\_on=[**"**CategoryID**"**])**

## Merging two tables row-wise

**Solution**

**t1=t\_product[:40]**

**t2=t\_product[40:]**

**pd.concat([t1,t2])**

**# you may need ignore\_index, if the row index overlaps**

**pd.concat([t1,t2], ignore\_index=True)**

## Merging two tables column-wise

**Solution**

**t1=t\_product[t\_product.columns[:3]]**

**t2=t\_product[t\_product.columns[3:]]**

**t1.columns**

**#output  
Index([ProductID, ProductName, CategoryID], dtype=object)**

**t2.columns**

**#output  
Index([UnitPrice, UnitsInStock, Discontinued], dtype=object)**

**t=pd.concat([t1,t2], axis=1)**

**len(t.columns)**

**#output**

**6**

# Transforming a Table

## Reshaping – melting and casting for table statistics

**Problem**

**A table contains observations for multiple objects, and one often has to perform various statistics on it, a useful framework for such problems is called melting and casting.**

**Solution**

**# syntax  
# melt(colNamesToCollapseIntoVariable)  
# pivot\_table(valueCol, rowCols, colCols, aggregationFunction)**

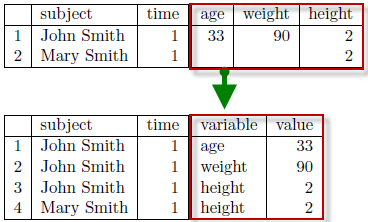
**# for two objects id = 1,2, we measure their x1 and x2 properties twice   
t = pd.DataFrame(np.array([[1,1,5,6], [1,2,3,5], [2,1,6,1], [2,2,2,4]]), columns=['id','time','x1','x2'])  
*# id time x1 x2  
# 1 1 5 6  
# 1 2 3 5  
# 2 1 6 1  
# 2 2 2 4*  
# first, melt a table into a tall-and-skinny table  
# using the combination of id and time as the key  
t2 = util.melt(t, ['x1','x2']);  
*#id time variable value  
# 1 1 x1 5  
# 1 1 x2 6  
# 1 2 x1 3  
# 1 2 x2 5  
# 2 1 x1 6  
# 2 1 x2 1  
# 2 2 x1 2  
# 2 2 x2 4*  
# casting the table, &average is a method to calculate mean  
# aggfunc=mean is the default  
#t2.pivot\_table('value', index='id', columns='variable')  
# in newer pandas version, rows replaced by index, cols replaced by columns  
t2.pivot\_table('value', index='id', columns='variable')  
*# id x1 x2  
# 1 4 5.5  
# 2 4 2.5***

**Discussion**

Hadley Wickham introduced the melting-and-casting framework for common problems in data reshaping and aggregation. The framework is implemented in the “Reshape” package in R and in Pandas

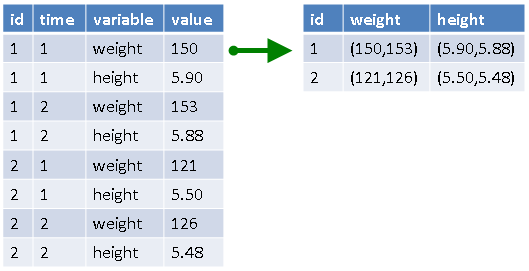
Melting basically unpivots a table.In this example, subjects (id = 1 and id = 2) were measured at time 1 and time 2 with two variables x1 and x2. Melting converts a short-and-wide table into a tall-and-skinny format, i.e., one specifies the columns for measurement variables. In this case, a unique combination of id-time is the id, and x1, x2 are two variables. Pandas actually uses stack() to do melt, stack will take all columns, that is why we wrote util.melt().

**As illustrated below (taken from the Reshape document**[[1]](#footnote-1)**), the idea of melting is to convert the typical database table into a tall-and-skinny fact table. The purpose of melting is to enable different groupings, i.e., casting.**



Casting is basically regrouping records into a contingency table. Here we choose **'id'** to be the row identifier and **'variable'** column contains data used to split **'value'** into different columns. As numerical values cannot be used as column names, this should be indicated in the third parameter, so that appropriate column names can be created. We expect to obtain a contingency table of id-by-x1,x2. There are probably multiple records share the same id-variable combination, therefore fall into the same destination cell, therefore these entries need to be aggregated using the supplied method. Pandas use pivot\_table() for cast.

Let us repeat the casting process with another example below, where each id is measured twice for their weight and height. To regroup, we first define what will be our rows, i.e., unique id (group by id). Then we define the new column should be taken from the “variable”, each unique value (“weight” and “height”) becomes a new column. Then we fill the “values” into corresponding cells in the result table, i.e., each cell contains an array of “values” that match the row id and column header. Last we apply an aggregation method, say average, to each cell and generate the final result. The contribution of melt() is to restructure the data in such a way, that one can group data by id, by time, by id-time, etc. pivot\_table() here is very similar to Excel’s pivot function.



For the product table, if one would like to calculate total cost of products in each category, use

t\_product['cost']=t\_product['UnitPrice']\*t\_product['UnitsInStock']

t\_product.pivot\_table('cost', index='CategoryID', aggfunc=sum)  
#output  
*CategoryID*

*1 12480.25*

*2 12023.55*

*3 10392.20  
...*

The first parameters indicates we would like to use cost column to fill the cells, each row is a unique CategoryID. Since we do not have a column that contains the new column names, we skip cols parameter. We specify sum as the aggfunc.

Let us look at another example, where we start with an employee salary table and try to calculate average salary for different groupings.

t = pd.DataFrame([

('Tom', 'male', 'IT', 65000),

('John', 'male', 'IT', 75000),

('Tom', 'male', 'IT', 65000),

('John', 'male', 'IT', 75000),

('Peter', 'male', 'HR', 85000),

('Mary', 'female', 'HR', 80000),

('Nancy', 'female', 'IT', 55000),

('Jack', 'male', 'IT', 88000),

('Susan', 'female', 'HR', 92000)

],

columns=['Name', 'Sex', 'Department', 'Salary'])  
# get a Department x Sex contingency table, get average salary across all four groups  
# Department defines the row, Sex defines the column, Salary fills the cells for average  
print(t.pivot\_table('Salary', index='Department', columns='Sex'))  
*Sex female male*

*Department*

*HR 86000 85000*

*IT 55000 73600*  
# get average salary for each department  
# Department defines the row, '(all)' is the column, Salary fills the cells for average  
print(t.pivot\_table('Salary', index='Department'))  
*Department*

*HR 85666.666667*

*IT 70500.000000*

*Name: Salary*  
# get average salary for each gender  
# Sex defines the row, '(all)' is the column, Salary fills the cells for average  
print(t.pivot\_table('Salary', index='Sex'))  
*Sex*

*female 75666.666667*

*male 75500.000000*

*Name: Salary*  
# get average salary for all records  
#print(t.pivot\_table('Salary')) #does not work

Print(t['Salary'].mean())  
*#output  
75555.5555555556*

Please also read stack(), unstack() function in Pandas.

## Grouping a table with aggregation functions

**Solution**

**# syntax**

**data=[]**

**for k,t\_v in t\_product.groupby('CategoryID'):**

**data.append([k, t\_v.UnitPrice.mean(), t\_v.UnitsInStock.mean()])**

**print(pd.DataFrame(data, columns=['CategoryID','AvgUnitPrice','AvgUnitsInStock']))**

**Discussion**

Group all rows based on their primary key columns (first parameter), for each group, we can apply a function. The function will be given a DataFrame object as input.

The following is an example of modifying individual dataframe objects, then concatenate them together into a new dataframe object.

**data=[]  
for k,t\_v in t.groupby('CategoryID'):  
 t\_v=t\_v.copy() # this line is no longer needed in the latest version  
 t\_v['RelativePricePct']=t\_v['UnitPrice']/t\_v['UnitPrice'].mean()  
 data.append(t\_v)  
t=pd.concat(data, ignore\_index=True)**

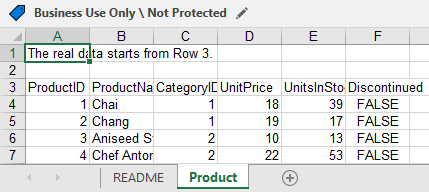
# Read Excel File

## Parse my\_product.xlsx

**Problem**

**Pandas package can only read Excel files, where the file contains just one spreadsheet and with only a data table inside. In real life, we need to handle more general Excel files, where there can be multiple spreadsheets (tabs) and the main data table can be embedded within the sheet, with other junk data.**

**In this chapter, we will show how to extract such a table. The example Excel file is called my\_product.xlsx (available from the code repository). The read data we are interested in on the second sheet, starting from the 3rd row.**



**We need to use our excel.py wrapper. If the example gives an error, you may need to install packages:**

pip install openpyxl, xlrd, xlwt

**Solution**

import pandas as pd  
import excel  
import util  
  
tables, names, headers, opts = excel.Excel.read('my\_product.xlsx')  
print(names)  
# ['README', 'Product']  
# We will read the 'Product' data sheet  
t=tables[util.index('Product', names)]  
t[:6].display()  
# Col1 Col2 Col3 Col4 Col5 Col6  
#-- -------------------------------- -------------------------------- ---------- --------- ------------ ------------  
# 0 The real data starts from Row 3.  
# 1  
# 2 ProductID ProductName CategoryID UnitPrice UnitsInStock Discontinued  
# 3 1 Chai 1 18 39 False  
# 4 2 Chang 1 19 17 False  
# 5 3 Aniseed Syrup 2 10 13 False

# The 3rd row is the header  
header=t.loc[2]  
# Real data starts from the 4th row  
t=t[3:]  
# fix the column header  
t.columns=header  
# reindex, so row index starts from 0, for convenience  
t.index=range(len(t))  
# Now we have the right data  
t[:3].display()  
# ProductID ProductName CategoryID UnitPrice UnitsInStock Discontinued  
#-- ----------- ------------- ------------ ----------- -------------- --------------  
# 0 1 Chai 1 18 39 False  
# 1 2 Chang 1 19 17 False  
# 2 3 Aniseed Syrup 2 10 13 False

1. http://had.co.nz/reshape/introduction.pdf [↑](#footnote-ref-1)