Lecture 5 - Data Wrangling: Clean, Transform, Merge, Reshape

Much of the programming work in data analysis and modeling is spent on data preparation: loading, cleaning, transforming, and rearranging. Sometimes the way that data is stored in files or databases is not the way you need it for a data processing application.

Combining and Merging Data Sets

Data contained in pandas objects can be combined together in a number of built-in ways: -pandas.merge connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database join operations. -pandas.concat glues or stacks together objects along an axis. -combine_first instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

Database-style DataFrame Merges

Merge or join operations combine data sets by linking rows using one or more keys. These operations are central to relational databases. The merge function in pandas is the main entry point for using these algorithms on your data. Let's start with a simple example:

In [1]:

```
import pandas as pd
```

In [2]:

```
from pandas import DataFrame, Series
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],'data1': range(7)})
df2 = DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
df1
```

Out[2]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

In [3]:

```
df2
```

Out[3]:

	key	data2
0	a	0
1	b	1
2	d	2

This is an example of a many-to-one merge situation; the data in df1 has multiple rows labeled a and b, whereas df2 has only one row for each value in the key column. Calling merge with these objects we obtain:

In [4]:

```
pd.merge(df1, df2)
```

Out[4]:

	key	data1	data2
0	b	0	1
1	b	1	1

	key	data1	data2
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

Note that I didn't specify which column to join on. If not specified, merge uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

In [5]:

```
pd.merge(df1, df2, on='key')
```

Out[5]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

If the column names are different in each object, you can specify them separately:

In [6]:

```
df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],'data1': range(7)})
df4 = DataFrame({'rkey': ['a', 'b', 'd'], 'data2': range(3)})
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

Out[6]:

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

You probably noticed that the 'c' and 'd' values and associated data are missing from the result. By default merge does an 'inner' join; the keys in the result are the intersec- tion. Other possible options are 'left', 'right', and 'outer'. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

In [7]:

```
pd.merge(df1, df2, how='outer')
```

Out[7]:

	key	data1	data2
0	b	0.0	1.0
1	b	1.0	1.0
2	b	6.0	1.0
3	a	2.0	0.0
4	a	4.0	0.0
5	a	5.0	0.0
6	c	3.0	NaN
7	d	NaN	2.0

In [9]:

```
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],'data1': range(6)})
df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],'data2': range(5)})
```

```
pd.merge(df1, df2, on='key', how='left')
```

Out[9]:

	key	data1	data2
0	b	0	1.0
1	b	0	3.0
2	b	1	1.0
3	b	1	3.0
4	a	2	0.0
5	a	2	2.0
6	c	3	NaN
7	a	4	0.0
8	a	4	2.0
9	b	5	1.0
10	b	5	3.0

To merge with multiple keys, pass a list of column names:

In [10]:

```
left = DataFrame({'key1': ['foo', 'foo', 'bar'],'key2': ['one', 'two', 'one'],'lval': [1, 2, 3]})
right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],'key2': ['one', 'one', 'one', 'two'],'rval': [4, 5, 6, 7]})

pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

Out[10]:

	key1	key2	lval	rval
0	foo	one	1.0	4.0
1	foo	one	1.0	5.0
2	foo	two	2.0	NaN
3	bar	one	3.0	6.0
4	bar	two	NaN	7.0

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the later section on renaming axis labels), merge has a suffixes option for specifying strings to append to overlapping names in the left and right DataFrame objects:

In [11]:

```
pd.merge(left, right, on='key1')
```

Out[11]:

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

In [14]:

```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

Out[14]:

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5

	key1	key2_left	lval	key2_right	rval
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

Merge function arguments

Argument	Description
`left`	DataFrame to be merged on the left side
`right`	DataFrame to be merged on the right side
`how`	One of `'inner'`, `'outer'`, `'left'` or `'right'`. `'inner'` by default
`on`	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in `left` and `right` as the join keys
`left_on`	Columns in `left` DataFrame to use as join keys
`right_on`	Analogous to `left_on` for `left` DataFrame
`left_index`	Use row index in `left` as its join key (or keys, if a MultiIndex)
`right_index`	Analogous to `left_index`
`sort`	Sort merged data lexicographically by join keys; `True` by default. Disable to get better performance in some cases on large datasets
`suffixes`	Tuple of string values to append to column names in case of overlap; defaults to `('_x', '_y')`. For example, if `'data'` in both DataFrame objects, would appear as `'data_x'` and `'data_y'` in result
`copy`	If `False`, avoid copying data into resulting data structure in some exceptional cases. By default always copies

Merging on Index

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass left_index=True or right_index=True (or both) to indicate that the index should be used as the merge key:

In [15]:

```
left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],'value': range(6)})
right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
left1
```

Out[15]:

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

In [16]:

```
right1
```

Out[16]:

	group_val
a	3.5

b	group_val

In [17]:

```
pd.merge(left1, right1, left_on='key', right_index=True)
```

Out[17]:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

In [18]:

```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

Out[18]:

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

Concatenating Along an Axis

Another kind of data combination operation is alternatively referred to as concatena- tion, binding, or stacking. NumPy has a concatenate function for doing this with raw NumPy arrays:

In [20]:

```
import numpy as np
arr = np.arange(12).reshape((3, 4))
arr
```

Out[20]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [21]:

```
np.concatenate([arr, arr], axis=1)
```

Out[21]:

```
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a number of additional things to think about: -If the objects are indexed differently on the other axes, should the collection of axes be unioned or intersected? -Do the groups need to be identifiable in the resulting object? -Does the concatenation axis matter at all? The concat function in pandas provides a consistent way to address each of these con- cerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

In [22]:

```
s1 = Series([0, 1], index=['a', 'b'])
s2 = Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = Series([5, 6], index=['f', 'g'])
pd.concat([s1, s2, s3])
```

```
pd.concat([s1, s2, s3])
```

Out[22]:

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

By default concat works along axis=0, producing another Series. If you pass axis=1, the result will instead be a DataFrame (axis=1 is the columns):

In [23]:

```
pd.concat([s1, s2, s3], axis=1)
```

Out[23]:

	0	1	2
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes. You can instead intersect them by passing join='inner':

In [24]:

```
s4 = pd.concat([s1 * 5, s3])
s4
```

Out[24]:

```
a    0
b    5
f    5
g    6
dtype: int64
```

In [25]:

```
pd.concat([s1, s4], axis=1)
```

Out[25]:

	0	1
a	0.0	0
b	1.0	5
f	NaN	5
g	NaN	6

In [26]:

```
pd.concat([s1, s4], axis=1, join='inner')
```

Out[26]:

	0	1
a	0	0
b	1	5

One issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keys argument:

In [28]:

```
result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
result
```

Out[28]:

```
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64
```

In [30]:

```
result.unstack()
```

Out[30]:

	a	b	f	g
one	0.0	1.0	NaN	NaN
two	0.0	1.0	NaN	NaN
three	NaN	NaN	5.0	6.0

In the case of combining Series along axis=1, the keys become the DataFrame column headers:

In [31]:

```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

Out[31]:

	one	two	three
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

The same logic extends to DataFrame objects:

In [32]:

```
df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],columns=['one', 'two'])
df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'], columns=['three', 'four'])
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

Out[32]:

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

If you pass a dict of objects instead of a list, the dict's keys will be used for the keys option:

In [33]:

```
pd.concat({'level1': df1, 'level2': df2}, axis=1)
```

Out[33]:

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Combining Data with Overlap

Another data combination situation can't be expressed as either a merge or concate- nation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy's where function, which expressed a vectorized if-else:

In [34]:

```
a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan], index=['f', 'e', 'd', 'c', 'b', 'a'])
b = Series(np.arange(len(a), dtype=np.float64), index=['f', 'e', 'd', 'c', 'b', 'a'])
b[-1] = np.nan
```

In [35]:

```
a
```

Out[35]:

```
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64
```

In [36]:

```
b
```

Out[36]:

```
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
dtype: float64
```

In [37]:

```
np.where(pd.isnull(a), b, a)
```

Out[37]:

```
array([0. , 2.5, 2. , 3.5, 4.5, nan])
```

Reshaping and Pivoting

There are a number of fundamental operations for rearranging tabular data. These are alternatingly referred to as reshape or pivot operations.

Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions: -stack: this "rotates" or pivots from the columns in the data to the rows -unstack: this pivots from the rows into the columns I'll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

In [39]:

```
data = DataFrame(np.arange(6).reshape((2, 3)),
                 index=pd.Index(['Ohio', 'Colorado'], name='state'),
                 columns=pd.Index(['one', 'two', 'three'], name='number'))
data
```

Out[39]:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

In [41]:

```
result = data.stack()
result
```

Out[41]:

```
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

In [42]:

```
result.unstack()
```

Out[42]:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

In [43]:

```
result.unstack(0)
```

Out[43]:

state	Ohio	Colorado
number		
one	0	3
two	1	4
three	2	5

In [44]:

```
result.unstack('state')
```

Out[44]:

state	Ohio	Colorado
number		
one	0	3
two	1	4
three	2	5

Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

In [45]:

```
s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = Series([4, 5, 6], index=['c', 'd', 'e'])
data2 = pd.concat([s1, s2], keys=['one', 'two'])
data2.unstack()
```

```
data2.unstack()
```

Out[45]:

	a	b	c	d	e
one	0.0	1.0	2.0	3.0	NaN
two	NaN	NaN	4.0	5.0	6.0

Stacking filters out missing data by default, so the operation is easily invertible:

In [46]:

```
data2.unstack().stack()
```

Out[46]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64
```

In [47]:

```
data2.unstack().stack(dropna=False)
```

Out[47]:

```
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

Data Transformation

Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

In [49]:

```
data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4, 'k2': [1, 1, 2, 3, 3, 4, 4]})
data
```

Out[49]:

	k1	k2
0	one	1
1	one	1
2	one	2
3	two	3
4	two	3
5	two	4
6	two	4

The DataFrame method duplicated returns a boolean Series indicating whether each row is a duplicate or not:

In [50]:

```
data.duplicated()
```

Out[50]:

```
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

Relatedly, drop_duplicates returns a DataFrame where the duplicated array is True:

In [51]:

```
data.drop_duplicates()
```

Out[51]:

	k1	k2
0	one	1
2	one	2
3	two	3
5	two	4

Both of these methods by default consider all of the columns; alternatively you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1' column:

In [53]:

```
data['v1'] = range(7)
data.drop_duplicates(['k1'])
```

Out[53]:

	k1	k2	v1
0	one	1	0
3	two	3	3

duplicated and drop_duplicates by default keep the first observed value combination. Passing take_last=True will return the last one:

In [56]:

```
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[56]:

	k1	k2	v1
1	one	1	1
2	one	2	2
4	two	3	4
6	two	4	6

Transforming Data Using a Function or Mapping

For many data sets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. Consider the following hypothetical data collected about some kinds of meat:

In [57]:

```
data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami','corned beef', 'Bacon', 'pastrami', 'honey ham','nova lox'],
                  'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

Out[57]:

	food	ounces
0	bacon	4.0

	food	ounces
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

Suppose you wanted to add a column indicating the type of animal that each food came from. Let's write down a mapping of each distinct meat type to the kind of animal:

In [58]:

```
meat_to_animal = {
'bacon': 'pig',
'pulled pork': 'pig',
'pastrami': 'cow',
'corned beef': 'cow',
'honey ham': 'pig',
'nova lox': 'salmon'
}
```

The map method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats above are capitalized and others are not. Thus, we also need to convert each value to lower case:

In [59]:

```
data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
data
```

Out[59]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

We could also have passed a function that does all the work:

In [60]:

```
data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Out[60]:

```
0      pig
1      pig
2      pig
3      cow
4      cow
5      pig
6      cow
7      pig
8    salmon
Name: food, dtype: object
```

Replacing Values

Filling in missing data with the fillna method can be thought of as a special case of more general value replacement. While map, as you've seen above, can be used to modify a subset of values in an object, replace provides a simpler and more flexible way to do so. Let's consider this Series:

In [61]:

```
data = Series([1., -999., 2., -999., -1000., 3.])
data
```

Out[61]:

```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series:

In [64]:

```
data.replace(-999, np.nan)
```

Out[64]:

```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

In [63]:

```
data.replace([-999, -1000], np.nan)
```

Out[63]:

```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

To use a different replacement for each value, pass a list of substitutes:

In [65]:

```
data.replace([-999, -1000], [np.nan, 0])
```

Out[65]:

```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

The argument passed can also be a dict:

In [66]:

```
data.replace({-999: np.nan, -1000: 0})
```

Out[66]:

```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects. The axes can also be modified in place without creating a new data structure. Here's a simple example:

In [67]:

```
data = DataFrame(np.arange(12).reshape((3, 4)),
      index=['Ohio', 'Colorado', 'New York'],columns=['one', 'two', 'three', 'four'])
```

In [68]:

```
data.index.map(str.upper)
```

Out[68]:

Index(['OHIO', 'COLORADO', 'NEW YORK'], dtype='object')

In [70]:

```
data.index = data.index.map(str.upper)
data
```

Out[70]:

	one	two	three	four
OHIO	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

In [71]:

```
data.rename(index=str.title, columns=str.upper)
```

Out[71]:

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

Notably, rename can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

In [72]:

```
data.rename(index={'OHIO': 'INDIANA'},columns={'three': 'peekaboo'})
```

Out[72]:

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Con- sider a DataFrame with some normally distributed data:

In [75]:

```
np.random.seed(12345)
data = DataFrame(np.random.randn(1000, 4))
data.describe()
```

Out[75]:

	0	1	2	3

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067684	0.067924	0.025598	-0.002298
std	0.998035	0.992106	1.006835	0.996794
min	-3.428254	-3.548824	-3.184377	-3.745356
25%	-0.774890	-0.591841	-0.641675	-0.644144
50%	-0.116401	0.101143	0.002073	-0.013611
75%	0.616366	0.780282	0.680391	0.654328
max	3.366626	2.653656	3.260383	3.927528

Suppose you wanted to find values in one of the columns exceeding three in magnitude:

In [76]:

```
col = data[3]
col[np.abs(col) > 3]
```

Out[76]:

```
97     3.927528
305   -3.399312
400   -3.745356
Name: 3, dtype: float64
```

To select all rows having a value exceeding 3 or -3, you can use the any method on a boolean DataFrame:

In [77]:

```
data[(np.abs(data) > 3).any(1)]
```

Out[77]:

	0	1	2	3
5	-0.539741	0.476985	3.248944	-1.021228
97	-0.774363	0.552936	0.106061	3.927528
102	-0.655054	-0.565230	3.176873	0.959533
305	-2.315555	0.457246	-0.025907	-3.399312
324	0.050188	1.951312	3.260383	0.963301
400	0.146326	0.508391	-0.196713	-3.745356
499	-0.293333	-0.242459	-3.056990	1.918403
523	-3.428254	-0.296336	-0.439938	-0.867165
586	0.275144	1.179227	-3.184377	1.369891
808	-0.362528	-3.548824	1.553205	-2.186301
900	3.366626	-2.372214	0.851010	1.332846

Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the numpy.random.permutation function. Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:

In [79]:

```
df = DataFrame(np.arange(5 * 4).reshape(5, 4))
sampler = np.random.permutation(5)
sampler
```

Out[79]:

```
array([1, 0, 2, 3, 4])
```

That array can then be used in ix-based indexing or the take function:

In [80]:

```
df
```

Out[80]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

In [81]:

```
df.take(sampler)
```

Out[81]:

	0	1	2	3
1	4	5	6	7
0	0	1	2	3
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

Additional https://datacarpentry.org/python-ecology-lesson/05-merging-data/index.html

In [ ]:

```

```