Python Libraries for Researchers

Research Computing Summer School 2019

Ian Percel

University of Calgary, Research Computing Services

May 28, 2019

Who is here?

- Who has programmed in Python before?
- Who has used NumPy before?
- Who is familiar with Linear Algebra?
- Who has used Pandas before?
- Who is familiar with Databases and Relational Algebra?

What is this talk about?

- Data structures are infrastructure
- `ndarray` and `DataFrame` are complex data structures
- The ability to blend linear algebra and relational algebra in a single family of structures brings opportunities and risk
- It is important to understand the philosophy of these structures. Once we do, many of the more complex functions become easy to understand.

Outline

Outline

Downloading this presentation

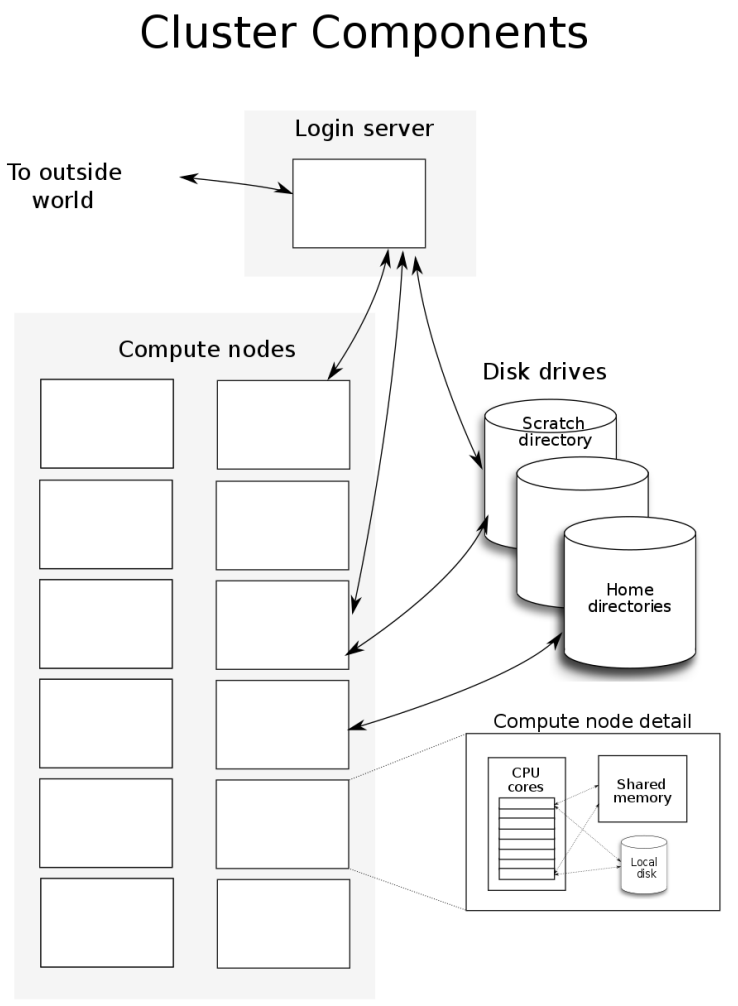
`https://westgrid.github.io/calgarySummerSchool2019/4-materials.html`

Right click on the Python Libraries for Reserchers:**Presentation** link and Save As/download to your computer

Downloading Data

- We will be working with US Census Data from the 5-year American Community Survey
- Specifically, we will be using the de-identified Public Use Microdata Sample (PUMS) data from 2013
- Point your browser at `https://www2.census.gov/acs2013_5yr/pums/` to see the relevant FTP directory
- Download `csv_hil.zip` to your personal computer (by right clicking and choosing Save As)

Cluster Architecture: where we will be working



Transferring Data to ARC on a Mac

- We will transfer the data set to your account using the `rsync` utility
- On a Mac: open Terminal
- From your Terminal run the following command

```
rsync -avv path/to/file/csv_hil.zip userName@arc.ucalgary.ca:"~"
```

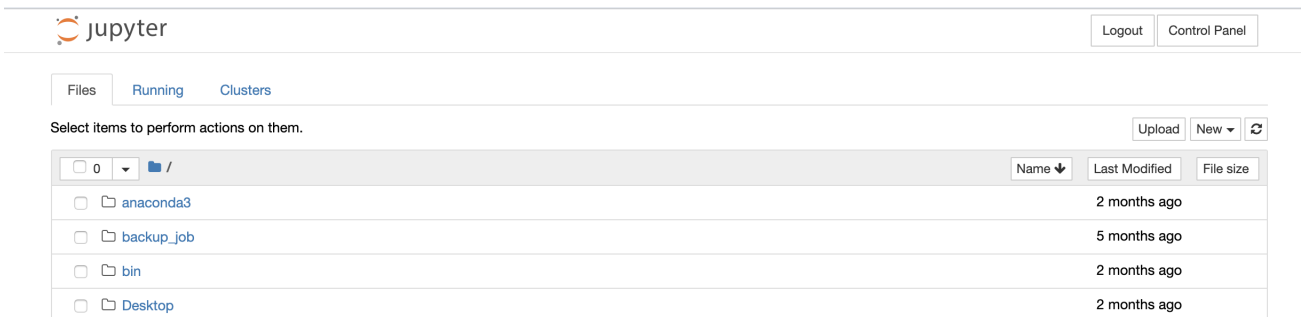
- path/to/file is the full path to the downloaded file
- on a mac desktop this would be ∼/Desktop/
- userName is your itUserName or guestUserName
- you will be prompted for a password, enter your ucalgary email password or the guest password that you have been given.
- If this is your first session signing in, you will be asked to confirm the certificate. Type yes and press enter.
- Once the transfer completes, enter the command: `ssh userName@arc.ucalgary.ca` and enter your password again

Transferring Data to ARC on a Windows PC

- On a Windows PC: open MobaXterm
- To connect an SSH session, the remote host=arc.ucalgary.ca, user name= your IT user Name or guest username
- You will be prompted for a password and will need to enter either your ucalgary email password or the guest password that you have been given
- If this is your first session signing in, you will be asked to confirm the certificate. Type yes and press enter
- An ASCII Art "ARC" welcome message should appear in the terminal.
- When the SSH Session connects an FTP window will appear on the left hand side. This can be used to upload the zip file graphically.
- Once the file has been uploaded, return to the prompt in your Moba terminal, type `unzip csv_hil.zip` and press enter

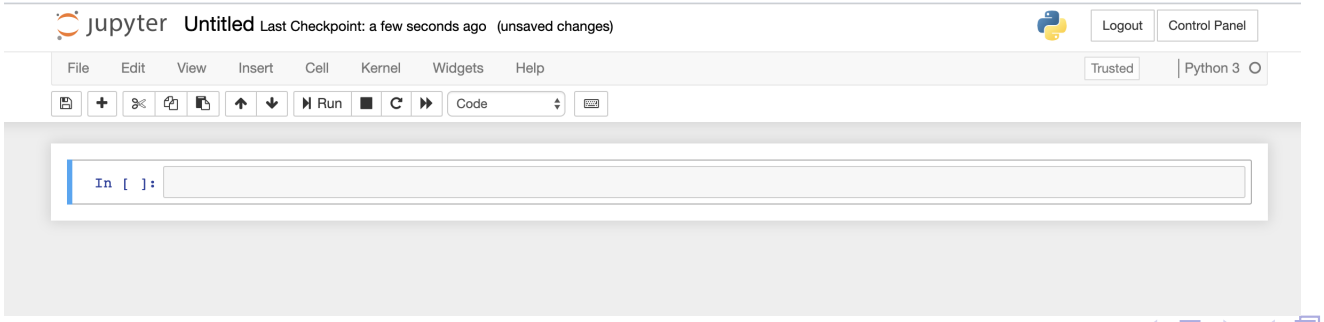
Jupyter Notebooks on ARC

- Why use Notebooks when custom installed environments are cleaner, faster, and more reliable? They're Prettier!
- `https://jupyter.ucalgary.ca:8000/hub/login`
- Use your itusername and email password to login
- Upload any data files that you need to use with the upload button
- Create a new notebook using New > Notebook: Python 3



Jupyter Notebooks on ARC

- Rename notebook by double-clicking on the work Untitled and changing it in the provided field and clicking the rename button at the bottom right of the dialogue
- To run python code, enter it in the text box / cell and press the run button (pressing enter will just create a newline) try out 3+5
- The result will be printed below the cell
- A new cell will be automatically be created below the cell that was just run



Where we are going

PUMS Data:

```
import pandas as pd
import numpy as np
from pandas import DataFrame,Series

basedf=pd.read_csv('ss13hil.csv')
#what are the columns?
print(list(basedf.columns))
```

```
['insp', 'RT', 'SERIALNO', 'DIVISION', 'PUMA00', 'PUMA10', 'REGION', 'ST', 'ADJHSG', 'ADJINC', 'WGTP',
'NP', 'TYPE', 'ACR', 'AGS', 'BATH', 'BDSP', 'BLD', 'BUS', 'CONP', 'ELEP', 'FS', 'FULP', 'GASP', 'HFL',
'MHP', 'MRGI', 'MRGP', 'MRGT', 'MRGX', 'REFR', 'RMSP', 'RNTM', 'RNTP', 'RWAT', 'RWATPR', 'SINK', 'SMP',
'STOV', 'TEL', 'TEN', 'TOIL', 'VACS', 'VALP', 'VEH', 'WATP', 'YBL', 'FES', 'FINCP', 'FPARC', 'GRNTP',
'HHL', 'HHT', 'HINCP', 'HUGCL', 'HUPAC', 'HUPAOC', 'HUPARC', 'KIT', 'LNGI', 'MULTG', 'MV', 'NOC', ...]
#plus 50 more real columns and 80 replication weights
```

```
#How many rows?
basedf.shape[0]
Out: 287799
```

Working in a graphical tool like Excel is out of the question. For more information:

https://www2.census.gov/programs-surveys/acs/tech_docs/pums/data_dict/PUMS_Data_Dictionary_2009-2013.pdf?#

Where we are going

How correlated is number of bedrooms (BDSP), the total number of rooms (RMSP) and number of persons listed (NP) by PUMA? We need to incorporate replication weights into our regression and repeat it for every PUMA. How do we do this in any reasonable amount of time?

```
basedf=pd.read_csv('ss13hil.csv', index_col='SERIALNO',
                usecols=['SERIALNO', 'PUMA00','WGTP', 'NP', 'BDSP', 'RMSP'], )
def f(base):
    subset=base.query('PUMA00==3515').dropna()
    W=np.diag(subset.query('PUMA00==3515')[['WGTP']].values.astype(np.float32).flatten())
    y=subset.query('PUMA00==3515')[['NP']].values.astype(np.float32).flatten()
    X=subset.query('PUMA00==3515')[['BDSP','RMSP']].values.astype(np.float32)
    XT=np.dot(W,X).T
    beta=np.dot(np.dot(np.linalg.inv(np.dot(XT,X)),XT),y)
    return beta
f(basedf)
[0.59527564, 0.06117937]
```

In 3515, 60% of the variation in NP can be explained by the variation in the number of bedrooms in the units sampled in contrast to 6% due to the number of rooms in the units sampled. (to describe the error in this statement we would need to look at the generalized uncertainty computed from all 80 of the replication weights)

By using the `apply` operator and eliminating the query clauses, this can be generalized to run separately on each

PUMA automatically and return a composite dataframe of the results.

ndarrays as Matrices

- NumPy can be thought of as a MATLAB like analysis tool
- If you can frame your problem in terms of linear operators (Matrix Algebra) then NumPy is your friend
- Some limitations:
- Single data-type per ndarray (all double precision floats [there are many possible number types] or all strings or all Booleans)
- Can be all Objects but this loses almost all of the power of NumPy
- Does not support query-like or JOIN-like operations that are familiar from working with other tables

What makes a good `ndarray`?

- Images
- Raster-type maps
- Coefficient matrices for finite-difference schemes
- Markov transition matrices
- Wiener Filters
- Quantized data for machine learning classification
- `word2vec` pre-processed text data
- Random vectors for use in Monte Carlo simulation

Making a Simple ndarray

```
import numpy as np
z1=np.zeros(4)
z1.shape
Out: (4)
z1.ndim
Out: 1
z2=np.zeros(4,4)
z2.shape
Out: (4,4)
z2.ndim
Out: 2
```

Making a Simple ndarray

```
a=np.eye(4)
a.dtype
Out: dtype('float64')
a=a.astype(np.bool)
a
Out: array([[ True, False, False, False],
       [False,  True, False, False],
       [False, False,  True, False],
       [False, False, False,  True]])
a.dtype
Out: dtype('bool')
```

Making an ndarray manually

```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
a.ndim
Out: 2
print(a)
Out:
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]

a[1]=[4,5,7]
print(a)
Out:
[[1 2 3]
 [4 5 7]
 [7 8 9]]

a[1,2]=13
print(a)
Out:
[[ 1  2  3]
 [ 4  5 13]
 [ 7  8  9]]

b=np.array([1,2,3])
b.ndim
Out: 1
print(b)
Out: [1, 2, 3]
```

Making an ndarray from randomly generated data

```
a=np.random.normal(0,25)
print(a)
Out: 15.936712317676044
type(a)
Out: float
b=np.random.normal(0,25,5)
print(b)
Out: [ 15.00295801   6.55752791 -21.31621132  17.15683612  -4.63542673]
type(b)
Out: <class 'numpy.ndarray'>
b.ndim
Out: 1
c=np.random.normal(0,25,(4,4))
print(c)
Out:
[[-53.66868452  11.20152646  10.40429457 -28.95341445]
 [-12.87286226  15.06092279  -1.20924852  -5.00258415]
 [-12.96715448  -4.11633735 -14.40398383  -8.23115077]
 [ 18.12714852 -18.11285746 -17.72416603 -14.91216281]]
c.ndim
Out: 2
```

Making an ndarray from a pandas DataFrame

```
import pandas as pd
import numpy as np

basedf=pd.read_csv('ss13hil.csv')
colList=[]
for n in range(1,81,1):
    colList.append('WGTP'+str(n))
simpledf=basedf[colList]
testArray=simpledf.iloc[0:80].values
print(type(testArray))
Out: <class 'numpy.ndarray'>
print(testArray.dtype)
Out: int64
print(testArray.ndim)
Out: 2
```

Problems 1

1. Generate a 100 x 100 identity matrix and set it to the variable `i`
2. Swap the first and second rows (Hint: create two 1D arrays from the first two vectors using `v1=i[0].copy()` and `v2=i[1].copy()`, use these to change the values of the rows of `i`) What will this new matrix do if you matrix multiply it against a vector?
3. Use `np.random.normal` to generate a 500 row and 2 column matrix of random samples from a normal distribution with mean 1000 and standard deviation of 20

Problems 2

1. Use `np.random.poisson` (`https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.poisson.html`) to generate a 1000 random samples from a poisson distribution with an expected interval of 10
2. Starting from `basedf=pd.read_csv('ss13hil.csv')` use the data dictionary to identify the column names associated with the 2010 PUMA microdata area code, the housing weight, the number of units in structure, the tenure, and the vacancy status. Create a list of these codes as strings and name it listCols. Set `df=basedf[listCols].copy()` and then attempt to turn this into an ndarray. What is the inferred datatype? What are the number of rows? what are the number of columns?

Element-wise Operations

- scalar multiplication (multiply every element by a number)
- element-wise reciprocal (replace every element by its inverse)
- element-wise exponentiation (raise every element to a power)
- unary universal functions (np.log, np.exp, np.isnan)
- binary universal functions (np.maximum, np.minimum)

Element-wise Operations

```
x=np.array([[1,2],[3,4]])
x*5
Out:
array([[ 5, 10],
       [15, 20]])

1/x
Out:
array([[1.        , 0.5       ],
       [0.33333333, 0.25      ]])

 x**(0.5)
 Out:
 array([[1.        , 1.41421356],
       [1.73205081, 2.        ]])
```

Element-wise Operations

```
np.log(x)
Out:
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])

x=x.astype(np.float64)
x[0,0]=np.nan
np.isnan(x)
Out:
array([[ True, False],
       [False, False]])

y=np.array([[1.5,2.5],[0,0]])
np.maximum(x,y)
Out:
array([[nan, 2.5],
       [3. , 4. ]])
```

Transposition

- Transposition is a special kind of rearranging of elements (reshaping in numpy terminology)
- Rows are switched into Columns
- This operation is essential to much of linear algebra

```
x=np.array([[1,2],[3,4]])
x.T
Out:
array([[1, 3],
       [2, 4]])
```

Matrix Addition and Other Binary Element-wise Operations

- Matrices of the same size can be combined through element-wise arithmetic (binary ufuncs)
- Addition and subtraction are the same as in usual matrix algebra

```
x=np.array([[1,2],[3,4]])
y=x*3

x+y
Out:
array([[ 4,  8],
       [12, 16]])

x-y
Out:
array([[-2, -4],
       [-6, -8]])
```

Matrix Addition and Other Binary Element-wise Operations

- Multiplication and division operate element-wise and so are not the same as matrix multiplication and factorization

```
x=np.array([[1,2],[3,4]])
y=x*3

x*y
Out:
array([[ 3, 12],
       [27, 48]])

x/y
Out:
array([[0.33333333, 0.33333333],
       [0.33333333, 0.33333333]])

y/x
Out:
array([[3., 3.],
       [3., 3.]])
```

Matrix Multiplication and Matrix Inversion

- To access the traditional matrix product in numpy, use the `np.dot()` function
- This multiplies rows of the first matrix by columns of the second
- This only works if the row length of the first matrix agrees with the column length of the second

```
x=np.array([[1,2],[3,4]])
y=np.array([[0,1],[1,0]])
np.dot(x,y)
Out:
array([[2, 1],
       [4, 3]])
```

Matrix Multiplication and Matrix Inversion

- `np.linalg` package contains the function `inv`, which numerically computes the matrix inverse of a square matrix

```
x=np.array([[1,2],[3,4]])
x_inv=np.linalg.inv(x)
np.dot(x,x_inv)
Out:
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
```

Matrix Multiplication and Matrix Inversion

- When applied to transposed 1D arrays it also can be used to calculate the inner product

```
x=np.array([[1,2],[3,4]])
y=np.array([0,1])
np.dot(y,x)
Out:
array([3, 4])

z=np.array([0,1]).T
np.dot(y,z)
Out:1

z2=np.array([1,0]).T
np.dot(y,z)
Out:0
```

Advanced Linear Algebra

- In addition to matrix inversion, `np.linalg` contains a wide range of advanced functionality
- `np.linalg.trace` computes the sum of diagonal elements (equivalently eigenvalues)
- `np.linalg.det` computes the determinant of a matrix
- `np.linalg.eig` computes the eigenvectors and eigenvalues of a square matrix
- `np.linalg.svd` computes the singular value decomposition of a matrix
- `np.linalg.solve` computes the solution to a linear system of equations
- `np.linal.lstsq` computes the least squares solution to an overdetermined linear system
- `np.linalg` is so useful that if you are working with numpy at any length it is probably worth importing in its own right

Problems 1

1. Use `np.random.normal` to generate a 10 by 10 matrix and save it to the variable `x`
2. The resulting matrix should have rank 10 with high probability. Confirm that it does with `np.linalg.matrix_rank(x)`. If it has Rank less than 10, re-generate the matrix until it does.
3. Compute the matrix inverse of your full rank matrix.
4. Take the matrix product of your random matrix with its inverse and save that to a new variable `id_approx`.
5. Create a new 10 by 10 identity matrix.
6. Subtract the new identity matrix from your `id_approx` and save that to a new variable called `inv_error`. (this is the error matrix relative to a perfect inversion)

Problems 2

1. Compute the 2-norm of the `inv_error` matrix using `np.linalg.norm` (see `https://docs.scipy.org/doc/numpy-1.12.0/reference/generated/numpy.linalg.norm.html`) The result should be a very very small number
2. Repeat this process by using `np.random.normal` to generate a matrix with twice as many rows as columns. Confirm that its rank is equal to its number of columns.
3. For this matrix produce a pseudoinverse using `np.linalg.pinv` and repeat the above analysis of how close the left matrix product is to an identity matrix. What about the right matrix product?

What is pandas?

- Pandas provides a SQL-like approach (that blends in elements of statistics and linear algebra) to analyzing tables of data
- DataFrames in R are very similar
- Differences from NumPy: Pandas allows multiple datatypes, includes richer querying and merging of datasets
- Pandas has been adopted as a de facto standard for input and vectorization across numerous disciplines
- Spatial Data Analysis, Machine Learning, Natural Language Processing, Visualization and Mapping

DataFrames and their Constructors

Pandas include both Series and DataFrame.
We will exclusively focus on DataFrames.
Constructor and data load functions have three key options:

- Column names and default column data types
- Indexes or row names
- NULL handling

```
import pandas as pd
from pandas import DataFrame
```

DataFrames and their Constructors (column-oriented)

```
dataDict={'ID':[1000,1212,1357,4908],
          'province':['AB', 'AB', 'BC', 'BC'],
          'income':[12000,43000,95000,79500]}
df=DataFrame(dataDict)
df.head()
```

Out[1]:

	ID	province	income
0	1000	AB	12000
1	1212	AB	43000
2	1357	BC	95000
3	4908	BC	79500

DataFrames and their Constructors (column-oriented)

```
dataDict={'ID':[1000,1212,1357,4908],
          'province':['AB', 'AB', 'BC', 'BC'],
          'income':[12000,43000,95000,79500]}
df=DataFrame(dataDict, index=dataDict['ID'])
df.head()
```

Out[2]:

	ID	province	income
1000	1000	AB	12000
1212	1212	AB	43000
1357	1357	BC	95000
4908	4908	BC	79500

DataFrames and their Constructors (column-oriented)

```
dataDict={'ID':[1000,1212,1357,4908],
          'province':['AB', 'AB', 'BC', 'BC'],
          'income':[12000,43000,95000,79500]}
df=DataFrame(dataDict,
             index=dataDict['ID'],
             columns=['province','income'])
df.head()
```

Out[4]:

	province	income
1000	AB	12000
1212	AB	43000
1357	BC	95000
4908	BC	79500

DataFrames and their Constructors (row-oriented)

```
r1={'ID':1000, 'province':'AB', 'income':12000}
r2={'ID':1212, 'province':'AB', 'income':43000}
r3={'ID':1357, 'province':'BC', 'income':95000}
r4={'ID':4908, 'province':'BC', 'income':79500}
dataList=[r1,r2,r3,r4]
index_temp=[]
for x in dataList:
     index_temp.append(x['ID'])
df=DataFrame(dataList,
             index=index_temp,
             columns=['province','income'])
df.head()
```

Out[6]:

	province	income
1000	AB	12000
1212	AB	43000
1357	BC	95000
4908	BC	79500

Loading data from a csv file

```
basedf=pd.read_csv('ss13hil.csv')
basedf[['SERIALNO', 'PUMA00', 'PUMA10', 'ST', 'ADJHSG', 'ADJINC'
        'WGTP', 'NP', 'TYPE', 'ACR', 'AGS', 'BATH', 'BDSP',
        'BLD', 'BUS', 'CONP', 'ELEP', 'FS', 'FULP']].head()
```

	SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
0	2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
1	2009000000075	1000	-9	17	1086032	1085467	6	1	1	1.0	NaN	1.0	3.0	1.0	2.0	0.0	200.0	2.0	2.0
2	2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
3	2009000000132	3510	-9	17	1086032	1085467	60	4	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	1.0	2.0	2.0
4	2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0

Loading data from a csv file

```
basedf=pd.read_csv('ss13hil.csv', index_col='SERIALNO',
          usecols=['SERIALNO', 'PUMA00', 'PUMA10', 'ST',
          'ADJHSG', 'ADJINC', 'WGTP', 'NP', 'TYPE', 'ACR',
          'AGS', 'BATH', 'BDSP', 'BLD', 'BUS', 'CONP', 'ELEP',
          'FS', 'FULP'])
basedf.head()
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000000075	1000	-9	17	1086032	1085467	6	1	1	1.0	NaN	1.0	3.0	1.0	2.0	0.0	200.0	2.0	2.0
2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
2009000000132	3510	-9	17	1086032	1085467	60	4	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	1.0	2.0	2.0
2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0

Accessing Data from a DataFrame

```
basedf.loc[[2009000000061]]

SERIALNO PUMA00 PUMA10 ST ADJHSG ADJINC WGTP NP...
2009000000061 3515 -9 17 1086032 1085467 36 0 ...

basedf.loc[[2009000000061],'NP']=1
basedf.loc[[2009000000061]]

SERIALNO   PUMA00 PUMA10 ST ADJHSG ADJINC WGTP NP...
2009000000061 3515 -9 17 1086032 1085467 36 1   ...

basedf.loc[[2009000000061,2009000000075]]
will capture 2 rows
```

Accessing Data from a DataFrame

```
basedf['newField']=22
basedf[['newField']].head()
```

SERIALNO	newField
2009000000061	22
2009000000075	22
2009000000108	22
2009000000132	22
2009000000150	22

Accessing Data from a DataFrame

```
basedf.iloc[:5]
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000000075	1000	-9	17	1086032	1085467	6	1	1	1.0	NaN	1.0	3.0	1.0	2.0	0.0	200.0	2.0	2.0
2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
2009000000132	3510	-9	17	1086032	1085467	60	4	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	1.0	2.0	2.0
2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0

Problems 1

1. Load the ss13hil.csv file to a DataFrame including only the columns: 'SERIALNO', 'PUMA00', 'PUMA10', 'HUPARC', 'KIT', 'LNGI', 'MULTG', 'MV', 'NOC', 'NPF', 'NPP', 'NR', 'NRC', 'OCPIP', 'WGTP' and force SERIALNO to be the index
2. Look up the 10th through 12th rows of the DataFrame (by index location)
3. Use the SERIALNO values obtained for the 10th through 12th rows and construct a filter by index value that returns the same rows
4. Select these rows again with only the columns 'HUPARC', 'KIT', 'LNGI', 'MULTG', 'MV'

Problems 2

1. Starting from the filtered DataFrame that you created above, assign this a new variable (say dataview1)
2. Take the same filtered DataFrame and assign it to a second variable (say df2) and this time use the `.copy()` command (usage: `df2=df1.loc[[something],[some cols]].copy()`)
3. Inspect your two "new" dataframes and confirm that they are the same as your original filtered data
4. We will now explore how they are different. Change a value in the basedf for one of your filtered columns and rows. Now examine your two derived DataFrames to see if your change propagated.
5. Conversely, make a different change to each of the two derived DataFrames and check the original to see if it propagated back.

This is the difference between deep and shallow copies. Shallow copies are made by reference only and so changes propagate while deepcopies are completely independent objects. The upside of deepcopies is that they have no entanglements or side effects related to other objects. They only change when you change them. The downside is that they require a separate memory allocation so they quickly become prohibitive when you are working with very large data sets.

Querying DataFrames

- The methods for data access that we have described so far don't lend themselves to searching for rows based on meaningful data
- In this section, we will describe complex methods for selecting subsets of a DataFrame
- We will focus on two methods: (1) `query` (2) Boolean Indexing
- The two methods are essentially equivalent in simple cases but Boolean Indexing can be much more general

query

- query takes a text string argument in the form (roughly) of a SQL WHERE clause
- Column names need to be referenced without quoting so suitable single-word names are needed
- https://pandas.pydata.org/pandas-docs/version/0.22/indexing.html#indexing-query

```
basedf.query('PUMA00==3515')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000002489	3515	-9	17	1086032	1085467	15	1	1	2.0	1.0	1.0	2.0	2.0	2.0	0.0	50.0	1.0	1.0
2009000002611	3515	-9	17	1086032	1085467	49	2	1	NaN	NaN	1.0	1.0	6.0	NaN	0.0	50.0	2.0	2.0
2009000002724	3515	-9	17	1086032	1085467	45	1	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	40.0	2.0	2.0
2009000006025	3515	-9	17	1086032	1085467	17	2	1	NaN	NaN	1.0	2.0	4.0	NaN	0.0	100.0	2.0	2.0
2009000009853	3515	-9	17	1086032	1085467	14	4	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	150.0	2.0	2.0
2009000010773	3515	-9	17	1086032	1085467	18	2	1	1.0	NaN	1.0	3.0	3.0	2.0	0.0	110.0	2.0	2.0

query

- The query functionality can work between fields.
- However, the only operators that I would rely on are (==, !=, <, >, <=, >=, &, |)
- query() is by default evaluated using the numexpr engine, which outperforms pure python on DataFrames of more than 200,000 rows

```
basedf.query('BDSP==NP')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0
2009000000225	3402	-9	17	1086032	1085467	19	2	1	NaN	NaN	1.0	2.0	9.0	NaN	0.0	1.0	2.0	2.0
2009000000256	600	-9	17	1086032	1085467	26	2	1	NaN	NaN	1.0	2.0	7.0	NaN	0.0	90.0	1.0	2.0
2009000000335	2700	-9	17	1086032	1085467	28	1	1	NaN	NaN	1.0	1.0	5.0	NaN	0.0	60.0	2.0	2.0
2009000000461	400	-9	17	1086032	1085467	43	4	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	100.0	1.0	2.0

query

- Arithmetic is possible although I can't speak to its efficiency
- The use of `in` and `not in` operators as well as `==['a','b',...]`, although parts of this will generally be evaluated using pure python

```
basedf.query('0<BDSP<NP & PUMA00==3515')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000002611	3515	-9	17	1086032	1085467	49	2	1	NaN	NaN	1.0	1.0	6.0	NaN	0.0	50.0	2.0	2.0
2009000013780	3515	-9	17	1086032	1085467	20	3	1	NaN	NaN	1.0	2.0	4.0	NaN	0.0	140.0	2.0	2.0
2009000022871	3515	-9	17	1086032	1085467	27	7	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	70.0	1.0	2.0
2009000024254	3515	-9	17	1086032	1085467	15	4	1	NaN	NaN	1.0	2.0	5.0	NaN	0.0	100.0	1.0	2.0
2009000030494	3515	-9	17	1086032	1085467	39	6	1	NaN	NaN	1.0	2.0	7.0	NaN	0.0	120.0	1.0	2.0
2009000046340	3515	-9	17	1086032	1085467	40	4	1	NaN	NaN	1.0	3.0	5.0	NaN	0.0	80.0	1.0	2.0

Boolean Indexing

- Boolean Indexing takes advantage of the fact the any Series derived from a dataframe keeps the same index as the dataframe
- It amounts to creating a mask that only lets certain values slip through while blocking others

```
mask=(basedf['PUMA00']==3515)
mask[0:5]
Out:
SERIALNO
2009000000061     True
2009000000075    False
2009000000108    False
2009000000132    False
2009000000150    False
Name: PUMA00, dtype: bool
```

Boolean Indexing

- The mask itself can be created separately and then applied

```
mask=(basedf['PUMA00']==3515)
basedf[mask].head()
```

SERIALNO	PUMA00	WGTP	NP	BDSP	RMSP
2009000000061	3515	36	0	2.0	5.0
2009000002489	3515	15	1	2.0	5.0
2009000002611	3515	49	2	1.0	4.0
2009000002724	3515	45	1	3.0	6.0
2009000006025	3515	17	2	2.0	5.0

Boolean Indexing

- The filtering logic and functions used to create the mask itself can be as complex as you like
- Anything you can write in Python can be implemented to create a mask as long as you keep the same index intact

```
def filt(x):
    if x==1:
        return True
    if x==2:
        return False
    if x>2:
        return True
mask=(basedf['PUMA00']==3515)&(basedf['BDSP'].map(filt))
basedf[mask].head()
```

SERIALNO	PUMA00	WGTP	NP	BDSP	RMSP
2009000002611	3515	49	2	1.0	4.0
2009000002724	3515	45	1	3.0	6.0
2009000009853	3515	14	4	4.0	6.0
2009000010773	3515	18	2	3.0	9.0
2009000012599	3515	60	4	8.0	10.0

Problems 1

1. Start from an import of the PUMS csv using the columns 'SERIALNO', 'PUMA00', 'BDSP', and 'NP' (as usual with SERIALNO as the index). load this to a variable named basedf
2. Use the command `basedf[['PUMA00']].drop_duplicates()` to produce a list of the unique PUMA regions from 2000.
3. Using the `query` command, select only those rows where the PUMA region number matches one specific one that you chose to work with from the previous step. How many records are returned? (look below the readout of sample rows to see a number) Repeat this for 4 different PUMAs and compare the counts returned for each. (what is the total?)

Problems 2

1. Write a query using the OR operator | that includes the records for all 4 of the PUMA regions that you examined above. Check that the count of rows agrees with the total you just obtained.
2. Extend your query using additional parentheses and the AND operator & (or `and` if you prefer) to select off those housing records that are in one of those 4 regions and have more than 4 people living in the residence.
3. Write a function that implements all of the PUMA logic and use the `.map(functionName)` operation demonstrated above to produce a Boolean Index mask for the PUMA
4. Write a second boolean index mask using the NP$>$ 4 condition and combine it with the mask you just created to produce a single mask that does the work of your earlier query.
5. Apply the mask to get the corresponding rows and check by quick inspection if it agrees with your earlier result (we will come back to how you can check this more precisely in the next section)

Combining DataFrames

- Just as linear algebraic operations are used to combine ndarrays, relational operators are used to combine DataFrames
- SQL operations like LEFT JOIN, INNER JOIN, UNION, INTERSECT, SELECT DISTINCT, GROUP BY and APPLY all have realizations as pandas operations
- In this section we will discuss the JOIN and UNION operations
- Pandas is meant to handle messy, inconsistent data and help in making it consistent
- Anywhere that a SQL operation would assume clean or unique data, Pandas operations are a little more complex because they are meant to allow for reality

General Reference: `https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html`

concat also known as UNION (and sometimes a cheap JOIN)

- concat provides a way to append data to the end of a DataFrame *or* to append columns to existing rows
- In its simplest form, it "adds rows" to the table, forcing agreement between the columns (by column name) where possible
- concat becomes subtle when using the option axis=1

concat as UNION

- concat accepts a list of DataFrames as its only required argument
- If the indexes are completely disjoint and the column names are the same, that outcome is identical to that of UNION from SQL

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'a':[7,8,9], 'b':[10,11,12]}, index=['u','v','w'])
pd.concat([df1,df2])
```

	a	b
x	1	4
y	2	5
z	3	6
u	7	10
v	8	11
w	9	12

concat as UNION for data with different columns

- If the indexes are completely disjoint and the column names are not the same, that outcome differs from that of UNION from SQL in that it is still permitted but the non-matching columns are all included and filled in with NULLS (or in pandas language `np.nan`)

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'a':[7,8,9], 'c':[10,11,12]}, index=['u','v','w'])
pd.concat([df1,df2])
```

	a	b	c
x	1	4.0	NaN
y	2	5.0	NaN
z	3	6.0	NaN
u	7	NaN	10.0
v	8	NaN	11.0
w	9	NaN	12.0

concat as JOIN

- If the indexes are overlapping and the column names are not the same *and* `axis=1` is used, that is identical to that of INNER JOIN from SQL

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'c':[7,8,9], 'd':[10,11,12]}, index=['x','y','z'])
pd.concat([df1,df2],axis=1)
```

	a	b	c	d
x	1	4	7	10
y	2	5	8	11
z	3	6	9	12

concat as ???

- For repeated indexes UNIONing on `axis=0` or repeated columns JOINing on `axis=1`, `concat` attempts to preserver as much data as possible
- The result can be a little bit bizzarre and getting back something unambiguous may mean reindexing or renaming columns
- It is better to avoid this situation altogether if possible

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'c':[7,8,9], 'd':[10,11,12]}, index=['x','y','z'])
pd.concat([df1,df2])
```

See Problem 1.

merge as JOIN

- merge is a holistic JOIN operator
- Like SQL JOINs, the options for using it are complex and take a great deal of practice to master
- We will focus on two options: on= and how=
- on determines the common column used to join the two together (a list of common columns can be specified
- note that the indexes are not preserved. To keep them .reset_index() before joining and then set the index from that column after or join on index (not covered here)

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'a':[1,2,3], 'c':[10,11,12]}, index=['u','v','w'])
pd.merge(df1,df2,on='a')
```

	a	b	c
0	1	4	10
1	2	5	11
2	3	6	12

merge as INNER JOIN

- how can be set to left, right, inner, or outer
- The left_on and right_on options specify the matching columns on the left and right join tables if they have different names
- Note that the default value of how is inner and this will filter out non-matching rows symmetrically

```
df1=DataFrame({'a1':[1,2,3], 'b':[4,5,6]})
df2=DataFrame({'a2':[1,2,7], 'c':[10,11,12]})
pd.merge(df1,df2,how='inner',left_on='a1',right_on='a2')
```

	a1	b	a2	c
0	1	4	1	10
1	2	5	2	11

merge as LEFT JOIN

- By setting `how` to left the merge preserves columns in the first listed DataFrame,
- This leaves `np.nan` in the columns from the second table when no matches are found

```
df1=DataFrame({'a1':[1,2,3], 'b':[4,5,6]})
df2=DataFrame({'a2':[1,2,7], 'c':[10,11,12]})
pd.merge(df1,df2,how='left',left_on='a1',right_on='a2')
```

	a1	b	a2	c
0	1	4	1.0	10.0
1	2	5	2.0	11.0
2	3	6	NaN	NaN

merge as FULL OUTER JOIN

- An outer join keeps non-matched rows from both tables, filling with np.nan as needed

```
df1=DataFrame({'a1':[1,2,3], 'b':[4,5,6]})
df2=DataFrame({'a2':[1,2,7], 'c':[10,11,12]})
pd.merge(df1,df2,how='outer',left_on='a1',right_on='a2')
```

	a1	b	a2	c
0	1.0	4.0	1.0	10.0
1	2.0	5.0	2.0	11.0
2	3.0	6.0	NaN	NaN
3	NaN	NaN	7.0	12.0

merge as multi-key JOIN

- By passing a list to each of the on options, the corresponding keys are matched sequentially
- In this case two rows are found to match if and only if the value of a1 matches a2 and key1 matches key2

```
df1=DataFrame({'a1':[1,2,3],'key1':['R','R','C'] ,'b':[4,5,6]})
df2=DataFrame({'a2':[1,2,7],'key2':['R','D','C'], 'c':[10,11,12]})
pd.merge(df1,df2,how='outer',left_on=['a1','key1'],right_on=['a2','key2'])
```

	a1	key1	b	a2	key2	c
0	1.0	R	4.0	1.0	R	10.0
1	2.0	R	5.0	NaN	NaN	NaN
2	3.0	C	6.0	NaN	NaN	NaN
3	NaN	NaN	NaN	2.0	D	11.0
4	NaN	NaN	NaN	7.0	C	12.0

Problems 1

1. Return to the slide "concat as ???" and run the code in it. Perform a similar experiment by altering the column names in df2 to match those in df1 and run it again. Finally, change the axis of concatenation to be axis=1 and run it a third time.
2. Describe for yourself how concat handles matching rows and columns under different circumstances.
3. Start from an import of the PUMS csv using the columns 'SERIALNO', 'PUMA00','WGTP', 'WGTP1', 'RMSP', 'BDSP', and 'NP' (as usual with SERIALNO as the index). load this to a variable named basedf
4. Make a list of 10 PUMA00 values
5. Create an empty DataFrame with the columns 'PUMA00','WGTP', 'WGTP1' with the statement
   `newdf=DataFrame('PUMA00':[],'WGTP':[], 'WGTP1':[] )`

Problems 2

1. Write a for loop that iterates over the list of PUMA00 values you just made and for each value queries basedf for the matching PUMA00 records, copies the resulting DataFrame and stores it to a temporary variable. Finally, the loop should concat the DataFrame to you just made by querying and copying to newdf.
2. When the loop finishes you will have assembled a DataFrame equivalent to querying for records belonging to any of those 5 PUMA regions. This is prototypical of assembling a dataframe through a sequence of complex operations that you append on a single output table.
3. Test this by querying the full table for those 5 PUMA regions using the OR operators and copying the result to a new temporary DataFrame then using merge to perform and inner join between the two tables on SERIALNO (HINT use reset_index() on each first to make the old index a column in each table)

If the resulting table has the same number of rows as the original two tables (and they have the same number of rows as themselves when deduplicating over SERIALNO) then they have the same set of records

Transforming Rows

- `map` applies a function to the content of every entry in a Series and can be used to systematically transform a single column of a DataFrame
- `applymap` works similarly but operates on every column of every row of a DataFrame
- `apply` is a profoundly general transformation function for breaking a Dataframe into sub-DataFrames, transforming each sub-DataFrame into new DataFrames, and finally re-assembling them

map for Transforming Columns

```
basedf['NP\_sq']=basedf['NP'].map(lambda x: x**2)
basedf['PUMA00\_str']=basedf['PUMA00'].map(lambda x: 'PUMA00:'+str(x))
```

Split-Apply-Combine as an overall strategy

- Similar to (but more general than) GROUP BY in SQL
- General tool for bulk changes
- The splitting step breaks data into groups using any column (including the row number) [1]
- This can be accomplished using `df.groupby('year')`

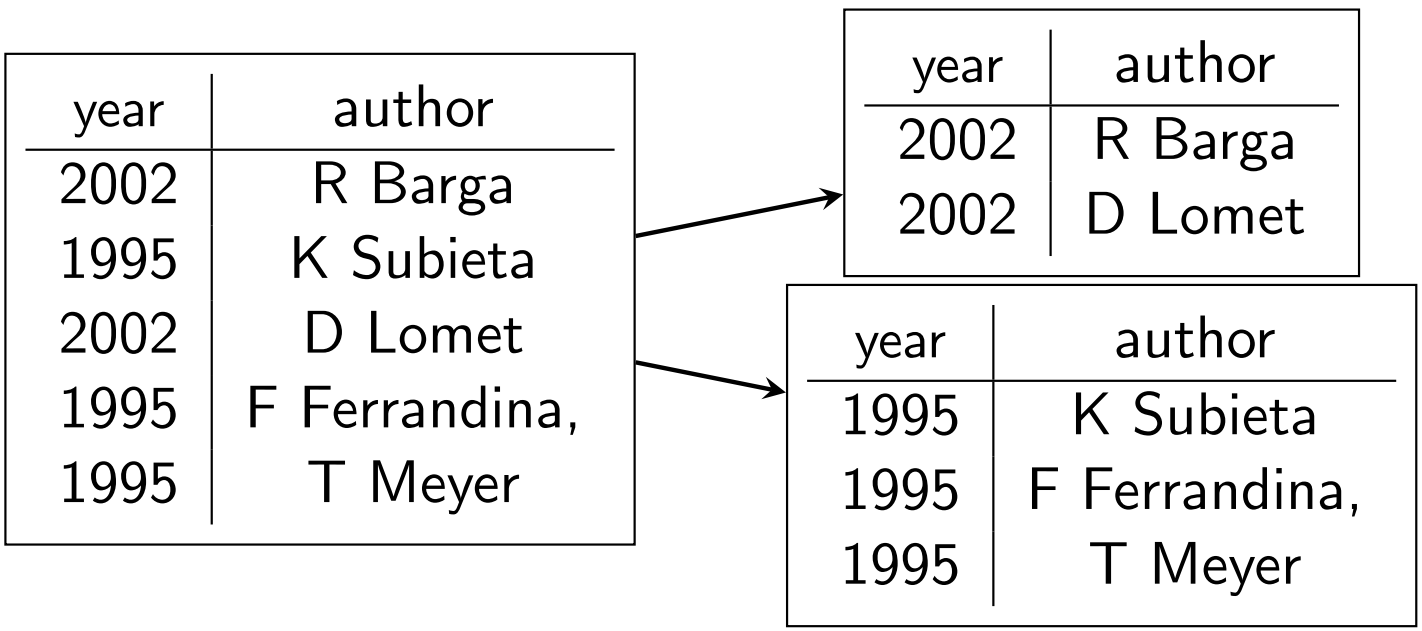


Figure 3: Table Split/Fork

Split-Apply-Combine in more detail

- Groups produced by the split can be individually transformed by an arbitrary function [1]
- This is the essence of Apply (the DataFrame extension of Map)
- The result is combined back into a single DataFrame

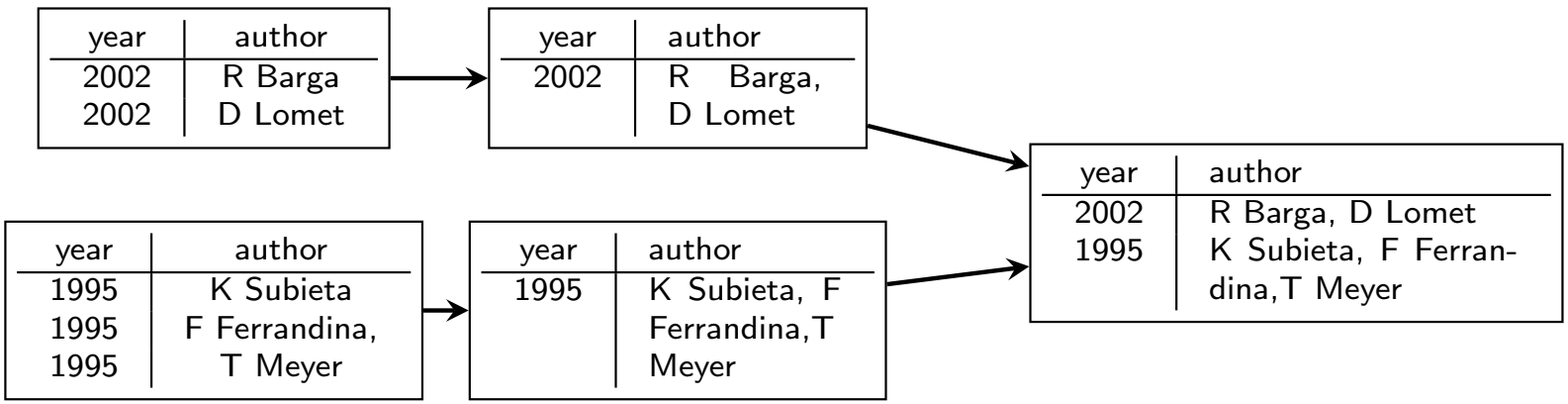


Figure 4: Table Apply + Combine for concatenation

All of this is performed by a single Python interpreter on a single machine.

apply in action

```
subdf=basedf.query('PUMA00==3515').copy()
def computeWeightedNP(x):
    x['weightedNP']=x['NP']*x['WGTP']
    return x
subdf=subdf.apply(computeWeightedNP, axis=1)
totals=subdf.sum()
totals['weightedNP']/totals['WGTP']
Out: 1.70737
```

apply as CROSS APPLY

```
def computeWeightedNP(x):
    x['weightedNP']=x['NP']*x['WGTP']
    #print(x)
    totals=x.sum()
    x['avgNP']=totals['weightedNP']/totals['WGTP']
    return x
subdf.groupby(['PUMA00']).apply(computeWeightedNP)
```

- Try out the column transformation examples from the "map as Column Transformation" slide
- Complete the example at the beginning of the presentation to automatically compute weighted regressions by mixing NumPy and Pandas and apply it for each PUMA

[1] Wes McKinney. *Python for Data Analysis*. O'Reilly, 2013.