Lecture 05: Welcome to Data Management

1. Remote Jupyter Notebooks
2. Saving your place in SSH: `screen`, `tmux`, `mosh`
3. Introduction to NumPy
4. NumPy Arrays and Operations
5. Introduction to Pandas
6. Working with Data Files in Pandas
7. DataFrame and Series Objects
8. Preview of Data Manipulation with Pandas
9. Data munging in the shell: `cut`, `tr`, & `sed`

Remote Jupyter Notebooks

- Run Jupyter on a remote server, access via web browser
- Benefits: More computing power, centralized data storage
- Use cases: Large datasets, ML model training, collaboration
- Setup: Install on server, configure for remote access, SSH tunneling

Introduction to SSH Session Management

- Importance: Maintains work across disconnections
- Common issues with standard SSH:
  - Lost work on network interruptions
  - Difficulty managing multiple tasks
- Solutions: `screen`, `tmux`, `mosh`

`screen`

- Terminal multiplexer: multiple virtual terminals in one session
- Persists sessions across disconnects
- Commands:
  - `screen`: Start new session
  - `Ctrl-a d`: Detach
  - `screen -r`: Reattach
  - `Ctrl-a c`: New window
  - `Ctrl-a n`: Next window

`screen` Example

NOTE: `screen` has no visual interface

```
~ on ☁️ christopher.seaman@gmail.com
❯ screen
```

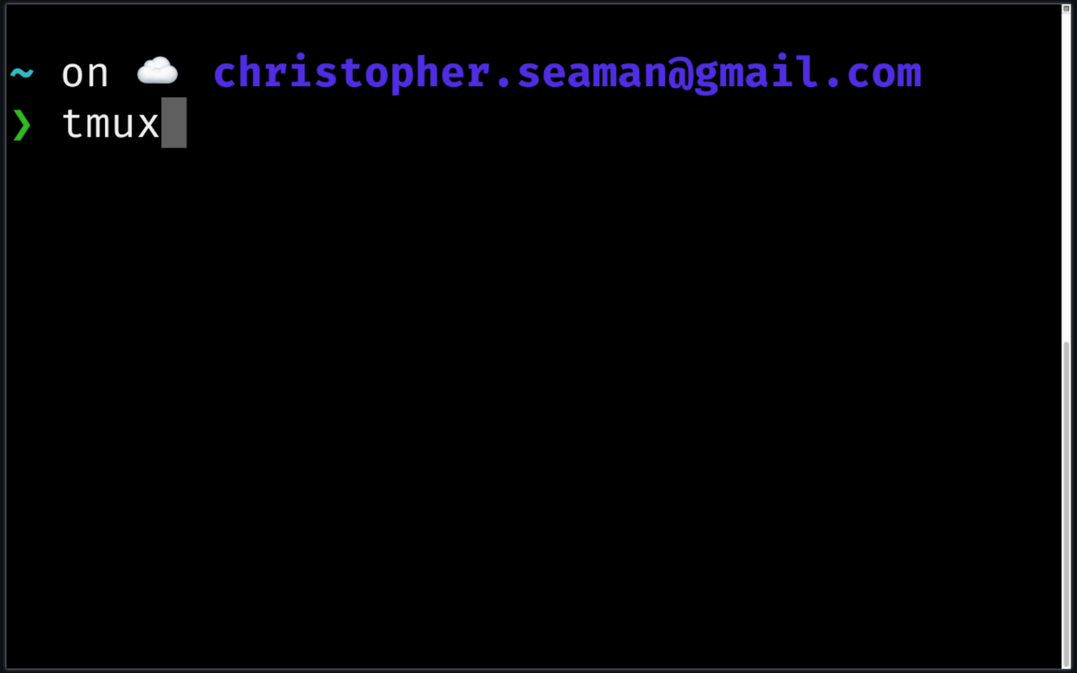
```
~ on ☁️ christopher.seaman@gmail.com
❯
```

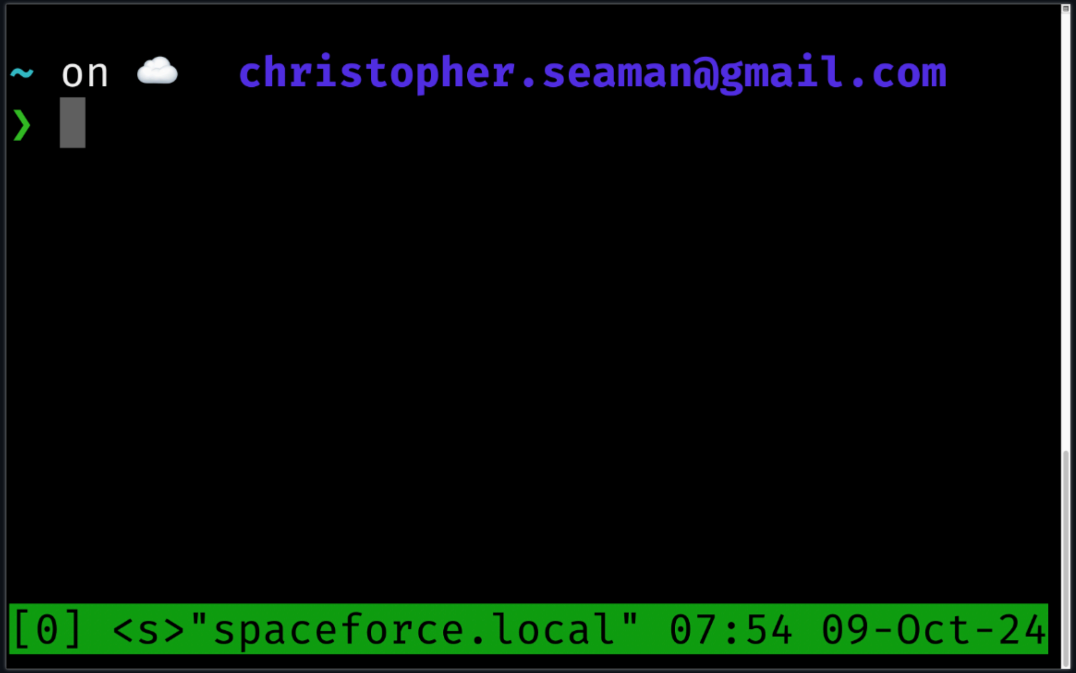
`tmux`

- Modern terminal multiplexer, highly customizable
- Commands:
  - `tmux`: Start or reattach to session
    - `tmux new -s NAME`: Start new session named `NAME`
    - `tmux attach -t NAME`: Reattach to session named `NAME`
    - `tmux ls`: List sessions
  - `Ctrl-b d`: Detach
  - `tmux attach`: Reattach
  - `Ctrl-b c`: New window
  - `Ctrl-b n`: Next window

`tmux` Example

NOTE: `tmux` has a status bar at the bottom





`mosh` (Mobile Shell)

- SSH replacement for unreliable networks
- Supports roaming and intermittent connectivity
- Pros: Works well on unreliable networks
- Cons: Requires server-side installation

LIVE DEMO!

What is NumPy?

- Numerical Python: fundamental for scientific computing
- Provides support for large, multi-dimensional arrays and matrices
- Efficient array operations (vectorized)
- Comprehensive mathematical functions
- Tools for integrating C/C++ and Fortran code
- Linear algebra, Fourier transform, and random number capabilities

The NumPy `ndarray` Object

- n-dimensional array
- Homogeneous data type
- Fixed size at creation

```python
import numpy as np
# Create a 1D array
a = np.array([1, 2, 3, 4, 5])

# Create a 2D array
b = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[1, 2, 3], # Equivalent
              [4, 5, 6]])

# Create a 3D array
c = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
c = np.array([[[1, 2],
               [3, 4]],
                       [[5, 6],
                        [7, 8]]])
```

Creating and Manipulating NumPy Arrays

```python
import numpy as np

# Creation
a = np.array([1, 2, 3, 4, 5])
b = np.zeros((3, 3))
c = np.ones((2, 2))
d = np.arange(0, 10, 2)

# Manipulation
e = a.reshape((5, 1))
f = b[1:, 1:]
g = c + 10
h = d * 2
```

Basic Array Operations

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise operations
c = a + b # [5, 7, 9]
d = a * b # [4, 10, 18]

# Matrix operations
e = np.dot(a, b) # 32
f = np.outer(a, b)
# [[4, 8, 12], [5, 10, 15], [6, 12, 18]]
```

NumPy Array Attributes

```python
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])

print(f"Shape: {arr.shape}")
print(f"Dimensions: {arr.ndim}")
print(f"Size: {arr.size}")
print(f"Data type: {arr.dtype}")
```

Output:

```
Shape: (2, 3)
Dimensions: 2
Size: 6
Data type: int64
```

NumPy Array Indexing and Slicing

Works like Python lists, but with additional features

```
import numpy as np

arr = np.array([[1,  2,  3,  4],
                [5,  6,  7,  8],
                [9, 10, 11, 12]])

print(arr[0, 2])            # Single element: 3
print(arr[1:, :2])          # Slice: [[5, 6], [9, 10]]
print(arr[[0, 2], [1, 3]])  # Advanced indexing: [2, 8]
# [0, 2] refers to the rows: row 0 and row 2
# The second list [1, 3] refers to:
#   - column 1 (of row 0)
#   - column 3 (of row 2)
```

Universal Functions (ufuncs)

- Fast element-wise array operations

Examples:

```
a = np.array([1, 4, 9])
b = np.sqrt(a)  # [1, 2, 3]
c = np.exp(a)   # [2.72, 54.60, 8103.08]
d = np.sin(a)   # [0.84, -0.76, 0.41]
```

Broadcasting

- Allows operations on arrays of different sizes
- NumPy's way of treating arrays with different shapes during arithmetic

```
# Not broadcasting
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
c = a * b  # Element-wise: [10, 40, 90, 160]

# Broadcasting
d = np.array([[1, 2, 3], [4, 5, 6]])
e = np.array([10, 20, 30])
f = d + e  # Broadcasting: [[11, 22, 33], [14, 25, 36]]
```

Broadcasting

$$\underset{(3,3)}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}} * \underset{(3,)\text{ or }(1,3)}{\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}} = \underset{(3,3)}{\begin{bmatrix} -1 & 0 & 3 \\ -4 & 0 & 6 \\ -7 & 0 & 9 \end{bmatrix}}$$

multiplying several columns at once

$$\underset{(3,3)}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}} / \underset{(3,1)}{\begin{bmatrix} 3 & 3 & 3 \\ 6 & 6 & 6 \\ 9 & 9 & 9 \end{bmatrix}} = \underset{(3,3)}{\begin{bmatrix} .3 & .7 & 1. \\ .6 & .8 & 1. \\ .8 & .9 & 1. \end{bmatrix}}$$

row-wise normalization

$$\underset{(3,)\text{ or }(1,3)}{\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}} * \underset{(3,1)}{\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}} = \underset{(3,3)}{\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}}$$

outer product

NumPy Array Operations

Generating sequences with `arange`

```
arr = np.arange(0, 10, 2)
```

Output:

```
[0 2 4 6 8]
```

NumPy Array Operations

Reshaping arrays:

```
arr = np.arange(12)
# arr = [ 0  1  2  3  4  5  6  7  8  9 10 11]

arr = arr.reshape(3, 4)
```

Output:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

NumPy Array Operations: `flatten()` & `ravel()`

- `flatten()` returns a copy of the array
- `ravel()` returns a flattened view of the array

```
arr = np.arange(12).reshape(3, 4)
# arr = [[ 0  1  2  3]
#         [ 4  5  6  7]
#         [ 8  9 10 11]]

# NOTE: arr will remain unchanged
flat_arr  = arr.flatten()
ravel_arr = arr.ravel()
```

Output:

```
# flat_arr is a new array with the flattened elements of arr
flat_arr = [ 0  1  2  3  4  5  6  7  8  9 10 11]

# ravel_arr is a view of the original array
ravel_arr = [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

NumPy Array Operations: Stacking

Vertical stacking:

```
arr1 = np.array([1, 2])
arr2 = np.array([3, 4])
stacked = np.vstack((arr1, arr2))
```

Output:

```
[[1 2]
 [3 4]]
```

Horizontal stacking

```
arr1 = np.array([1, 2])
arr2 = np.array([3, 4])

hstacked = np.hstack((arr1, arr2))
```

Output:

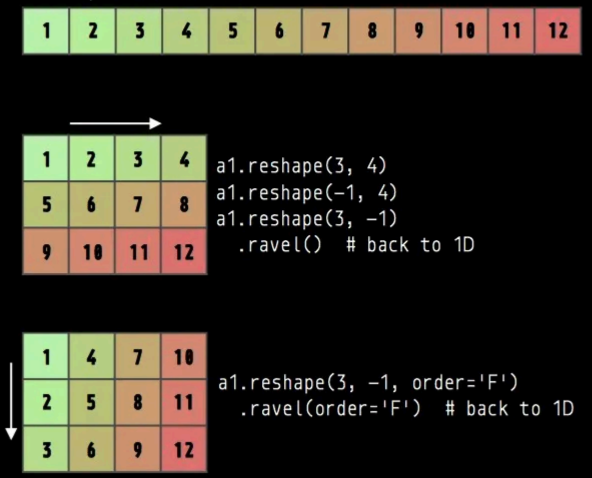
```
[1 2 3 4]
```

reshape & ravel

```
a1 = np.arange(1, 13)
```

```
a1.reshape(3, 4)
a1.reshape(-1, 4)
a1.reshape(3, -1)
  .ravel()  # back to 1D
```

```
a1.reshape(3, -1, order='F')
  .ravel(order='F')  # back to 1D
```



3D array from 2D arrays

```
a1 = np.arange(1, 13).reshape(3, 4)
a2 = np.arange(13, 25).reshape(3, -1)
```

```
# stack along axis 0
a3_0 = np.stack((a1, a2))
a3_0.shape: (2, 3, 4)
```

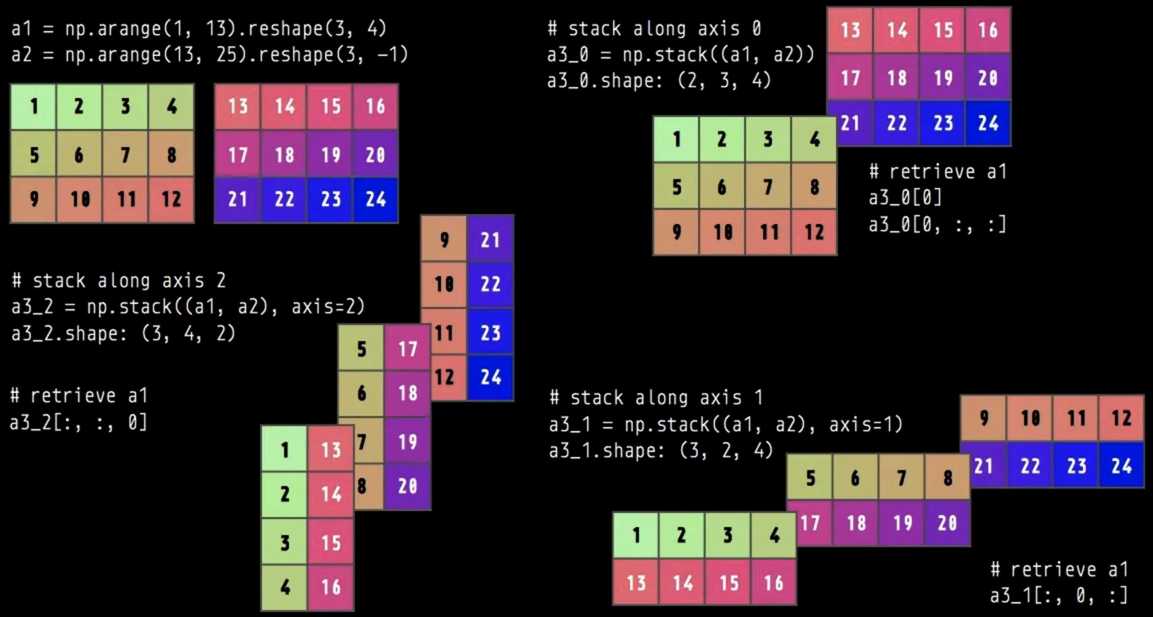
```
# retrieve a1
a3_0[0]
a3_0[0, :, :]
```

```
# stack along axis 2
a3_2 = np.stack((a1, a2), axis=2)
a3_2.shape: (3, 4, 2)
```

```
# retrieve a1
a3_2[:, :, 0]
```

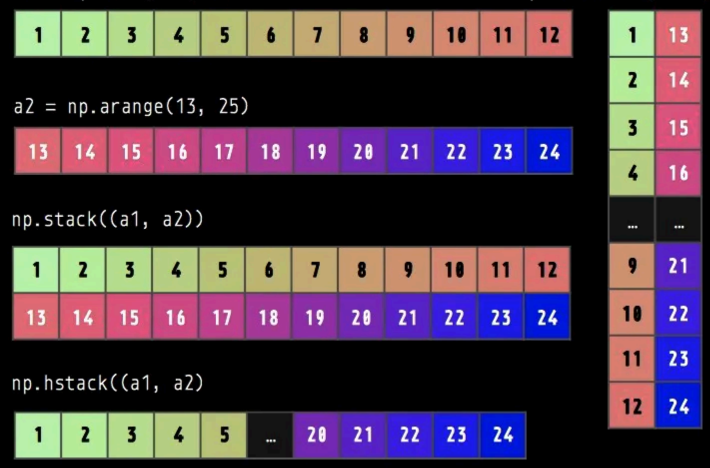
```
# stack along axis 1
a3_1 = np.stack((a1, a2), axis=1)
a3_1.shape: (3, 2, 4)
```

```
# retrieve a1
a3_1[:, 0, :]
```



stack

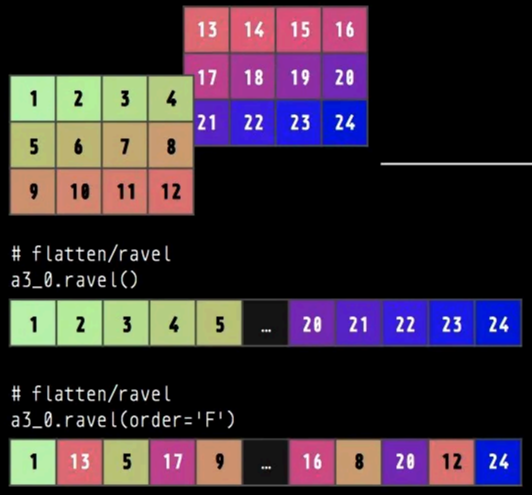
```
a1 = np.arange(1, 13)
a2 = np.arange(13, 25)
np.stack((a1, a2))
np.hstack((a1, a2))
np.stack((a1, a2), axis=1)
```



flatten 3D array

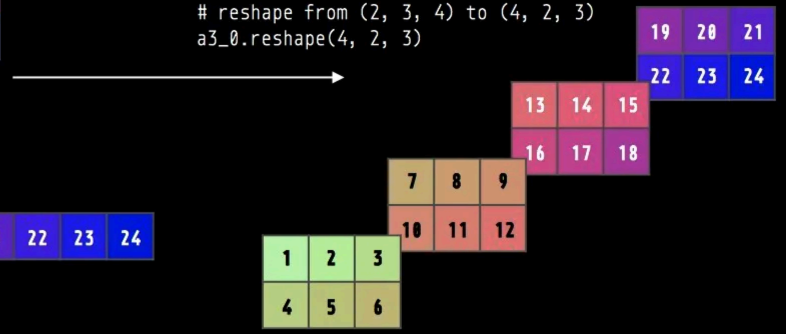
```
# flatten/ravel
a3_0.ravel()
```

```
# flatten/ravel
a3_0.ravel(order='F')
```



reshape 3D array

```
# reshape from (2, 3, 4) to (4, 2, 3)
a3_0.reshape(4, 2, 3)
```



LIVE DEMO!

What is `pandas`?

- Python Data Analysis Library
- Built on top of NumPy
- Provides high-performance, easy-to-use data structures and tools
- Designed for working with labeled and relational data
- Key data structures: Series (1D) and DataFrame (2D)

Pandas Series

- 1-dimensional labeled array
- Can hold data of any type

```python
import pandas as pd

s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
```

Output:

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Pandas DataFrame

- 2-dimensional labeled data structure
- Like a spreadsheet or SQL table

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data)
print(df)
```

Output:

```
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```

Creating Series and DataFrames

```python
# Series from dictionary
d = {'a': 1, 'b': 2, 'c': 3}
series = pd.Series(d)

# DataFrame from list of dictionaries
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)

print(series)
print("\n")
print(df)
```

Reading Data Files in Pandas

```python
# CSV
df_csv = pd.read_csv('filename.csv')

# JSON
df_json = pd.read_json('filename.json')

# Excel
df_excel = pd.read_excel('filename.xlsx', sheet_name='Sheet1')

# Handling data types
df_types = pd.read_csv('filename.csv',
                       dtype={'column1': str, 'column2': float})
```

Writing Data Files in Pandas

```python
# CSV
df.to_csv('output.csv', index=False)

# JSON
df.to_json('output.json')

# Excel
df.to_excel('output.xlsx', sheet_name='Sheet1')
```

Dealing with Missing Data

```
# Checking for missing values
print(df.isnull().sum())

# Dropping missing values
df_clean = df.dropna()

# Filling missing values
df_filled = df.fillna(value={'column1': 0, 'column2': 'Unknown'})
```

Basic DataFrame Operations

```python
# Viewing data
print(df.head())
print(df.tail())

# Information about DataFrame
print(df.info())
print(df.describe())

# Selecting columns
print(df['column_name'])
print(df[['column1', 'column2']])
```

Accessing Data in DataFrames

```
# Using loc for label-based indexing
print(df.loc['row_label', 'column_label'])

# Using iloc for integer-based indexing
print(df.iloc[0, 2])

# Boolean indexing
print(df[df['column'] > 5])
```

Pandas Data Selection and Filtering

```
import pandas as pd

df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': ['a', 'b', 'c', 'd'],
    'C': [True, False, True, False]
})

# Select single column
print(df['A'])

# Select multiple columns
print(df[['A', 'B']])

# Filter rows
print(df[df['A'] > 2])

# Combine selection and filtering
print(df.loc[df['C'], 'B'])
```

Pandas Basic Data Analysis

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50],
    'C': ['x', 'y', 'z', 'x', 'y']
})

# Basic statistics
print(df.describe())

# Value counts
print(df['C'].value_counts())

# Correlation
print(df.corr())

# Groupby and aggregate
print(df.groupby('C').mean())
```

LIVE DEMO

When to Use NumPy with Pandas

- NumPy and Pandas are often used together
- Pandas is built on top of NumPy
- Use NumPy when:
    1. You need low-level, fast array operations
    2. You're working with homogeneous numerical data
    3. You require advanced linear algebra or Fourier transforms
- Use Pandas when:
    1. You have labeled data or mixed data types
    2. You need SQL-like operations (groupby, join, etc.)
    3. You're handling time series data

NumPy and Pandas: Basic Interoperability

```python
import numpy as np
import pandas as pd

# Create a NumPy array
np_array = np.array([1, 2, 3, 4, 5])

# Convert NumPy array to Pandas Series
pd_series = pd.Series(np_array)

print("NumPy array:", np_array)
print("Pandas Series:", pd_series)

# Convert Pandas Series back to NumPy array
np_array_again = pd_series.to_numpy()

print("Back to NumPy:", np_array_again)
```

NumPy Operations on Pandas DataFrames

```python
import numpy as np
import pandas as pd

# Create a Pandas DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

# Use NumPy function on DataFrame
print("Original DataFrame:")
print(df)

print("\nNumPy square root:")
print(np.sqrt(df))

print("\nNumPy sum along columns:")
print(np.sum(df, axis=0))

print("\nNumPy mean along rows:")
print(np.mean(df, axis=1))
```

Advanced NumPy-Pandas Integration

```python
import numpy as np
import pandas as pd

# Create a Pandas DataFrame
df = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

print("Original DataFrame:")
print(df)

# Use NumPy to create a boolean mask
mask = np.abs(df) > 1

print("\nBoolean mask (absolute values > 1):")
print(mask)

# Apply the mask to the DataFrame
filtered_df = df[mask]

print("\nFiltered DataFrame:")
print(filtered_df)

# Use NumPy's where function with Pandas
df_modified = df.where(mask, other=0)

print("\nModified DataFrame (values <=1 replaced with 0):")
print(df_modified)
```

LIVE DEMO

Data Munging in the Shell

- Unix command-line tools for text processing
- Useful for quick data transformations
- Can be combined with pipes for complex operations

cut: Extracting Columns

- Selects specific columns from tabular data
- Usage: `cut OPTION... [FILE]...`

Examples:

```
# Extract 1st and 3rd columns from CSV
cut -d',' -f1,3 data.csv

# Extract characters 5-10 from each line
cut -c5-10 data.txt
```

tr: Translating Characters

- Transforms or deletes characters
- Usage: `tr [OPTION]... SET1 [SET2]`

Examples:

```
# Convert lowercase to uppercase
cat file.txt | tr 'a-z' 'A-Z'

# Delete all digits
echo "hello123world" | tr -d '0-9'

# Squeeze repeated characters
echo "hello    world" | tr -s ' '
```

sed: Stream Editor

- Powerful text transformation tool
- Can search, find and replace, insert, and delete

Examples:

```
# Replace first occurrence of 'old' with 'new'
sed 's/old/new/' file.txt

# Replace all occurrences
sed 's/old/new/g' file.txt

# Delete lines containing 'pattern'
sed '/pattern/d' file.txt

# Insert 'text' at beginning of each line
sed 's/^/text /' file.txt
```

Regular Expressions: Basics

- Powerful pattern matching tool
- Used with grep, sed, and many programming languages
- Test regex patterns online: https://regex101.com/
- Common regex elements:
    - `.` : Any single character
    - `*` : Zero or more of the previous character
    - `+` : One or more of the previous character
    - `?` : Zero or one of the previous character
    - `^` : Start of line
    - `$` : End of line
    - `[ ]` : Character class (match any character inside)
    - `[^]` : Negated character class (match any character not inside)

grep with Regular Expressions

```
# Match lines starting with "Error"
grep "^Error" logfile.txt

# Match email addresses
grep "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}" contacts.txt

# Match phone numbers (simple format)
grep "\d{3}-\d{3}-\d{4}" phonebook.txt

# Match words with at least 3 vowels
grep -E "[aeiou].*[aeiou].*[aeiou]" dictionary.txt
```

sed with Regular Expressions

```
# Replace "color" with "colour" (first occurrence on each line)
sed 's/color/colour/' document.txt

# Replace all occurrences of "color" with "colour"
sed 's/color/colour/g' document.txt

# Remove lines containing "DEBUG"
sed '/DEBUG/d' logfile.txt

# Add a prefix to lines starting with a number
sed 's/^[0-9]/PREFIX: &/' data.txt

# Wrap words in quotes
sed 's/\b\w+\b/"&"/g' text.txt
```

Combining Shell Commands

- Use pipes `|` to chain commands
- Create powerful data processing pipelines

Example:

```
# Extract 2nd column, convert to uppercase,
# replace spaces with underscores
cat data.csv | cut -d',' -f2 | tr 'a-z' 'A-Z' | tr ' ' '_'
```

cut: More Examples

```
# Extract fields 2-4 from a tab-separated file
cut -f2-4 data.tsv

# Use a custom delimiter
cut -d':' -f1,3 /etc/passwd

# Extract bytes 10-20 from each line
cut -b10-20 binary_data.bin
```

tr: Additional Use Cases

```
# Replace newlines with spaces
cat multiline.txt | tr '\n' ' '

# Remove all non-printable characters
cat file.txt | tr -cd '[:print:]'

# Translate multiple characters
echo "hello 123" | tr 'elo' 'EOL'
```

sed: Advanced Examples

```
# Add line numbers
sed = file.txt | sed 'N;s/\n/\t/'

# Remove empty lines
sed '/^$/d' file.txt

# Replace text between two patterns
sed '/start/,/end/c\Replacement text' file.txt

# Append text after a matching line
sed '/pattern/a\Appended text' file.txt
```

Advanced Regular Expression Examples

```
# Extract all IPv4 addresses using grep
grep -E '\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b' network_log.txt

# Format dates (MM/DD/YYYY to YYYY-MM-DD) using sed
sed -E 's,([0-9]{2})/([0-9]{2})/([0-9]{4}),\3-\1-\2,g' dates.txt

# Extract URLs from a file using grep
grep -Eo '(http|https)://[^[:space:]]+' webpage.html

# Remove HTML tags using sed
sed -E 's/<[^>]+>//g' webpage.html
```

Combining Shell Commands: Complex Example

```
# Process a CSV file:
# 1. Extract columns 2 and 4
# 2. Convert to uppercase
# 3. Replace commas with tabs
# 4. Sort numerically on the second field
# 5. Take the top 5 results
# 6. Save to a new file

cat data.csv | cut -d',' -f2,4 | tr '[:lower:]' '[:upper:]' | \
tr ',' '\t' | sort -k2 -n | head -n 5 > top_5_results.tsv
```

LIVE DEMO