Python Programming for Data Science

Week 40, Friday

Jupyter Notebooks

Pandas

Handin 4 - general comments

- No need to sort the words from file1 in the binary search question
- You can define local variables inside your constructor (not everything has to be an attribute)

```
class MyClass:
    def __init__(self, text):

        # There is no need to define temporary
        # variables as attributes. Instead of:
        self.text_stripped = text
        # you can just do:
        text_stripped = text

        self.wordlist = text_stripped.split()
```

Handin 4: it matters what you put in your inner loop (1)

```
def wordfile_differences_linear_search(filename1,
filename2):
    '''Find differences between two one-word-per-line
files using linear search'''

    word_list1 = wordfile_to_list(filename1)
    word_list2 = wordfile_to_list(filename2)

    ids = []
    for i, word1 in enumerate(word_list1):
        for j, word2 in enumerate(word_list2):
            if word1 == word2:
                break
            # Checking if this is the last index
            # If so, register as not found
            elif j == len(word_list2):
                ids.append(word1)

    return ids
```

This version takes 763 seconds on my computer.

Handin 4: it matters what you put in your inner loop (2)

```
def wordfile_differences_linear_search(filename1,
filename2):
    '''Find differences between two one-word-per-line
files using linear search'''

    word_list1 = wordfile_to_list(filename1)
    word_list2 = wordfile_to_list(filename2)

    ids = []
    for i, word1 in enumerate(word_list1):
        match = False
        for j, word2 in enumerate(word_list2):
            if word1 == word2:
                match = True
                break
        # Only checking once after inner loop
        if not match:
            ids.append(word1)

    return ids
```

This version takes 358 seconds on my computer.
The reason this version is faster is that we are only doing half as many operations in the inner loop.

Handin 4: it matters what you put in your inner loop (3)

```
def wordfile_differences_linear_search(filename1,
filename2):
    '''Find differences between two one-word-per-line
files using linear search'''

    word_list1 = wordfile_to_list(filename1)
    word_list2 = wordfile_to_list(filename2)

    ids = []
    for word1 in word_list1:
        # Linear search using "in" operator
        if not word1 in word_list2:
            ids.append(word)

    return ids
```

This version takes 58 seconds on my computer.

The reason that this version is faster is the same as that for Numpy: the inner loop is now delegated to a faster underlying implementation (in C).

Before we start: Install jupyter

To use the features described in these slides, start by installing the `jupyter` extension to VS Code:

`View` → `Extensions`. Search for "jupyter", and then click install

Or click

in the vertical menu bar on the left

An alternative solution: Google Colab

If you cannot get Jupyter notebooks to work on your own machine, you can always use Google Colab **colab.research.google.com** provides free notebook functionality, hosted on Googles servers.

- It is similar to jupyter notebooks (it builds on top of it)
- It provides free GPU resources
- It allows multiple users to work on the same notebook

You can download your jupyter notebook back to your own computer using `File -> Download -> Download .ipynb`
If you need to install modules in Google Colab, use `!pip`

```
!pip install pandas
```

The python console

Python has an interactive console

You can start it by typing `python` in a VS Code terminal window.

```
$ python
Python 3.11.5 (main, Sep 11 2023, 08:19:27) [Clang 14.0.6
] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

To exit the console, press `ctrl+d`

Python console: evaluation

In the console, Python will print out the result of any expression you enter
Enter a number:

```
>>> 10
10
```

Python returns the same number back.

Python console: evaluation (2)

Enter a sum:

```
>>> 10+20
30
```

Python calculates the sum.

Python console: evaluation (3)

Or any other Python expression:

```
>>> len([1,2,3,4])
4
>>> "1 2 3 4".split()
['1', '2', '3', '4']
```

Python console: what's it good for?

The console is a nice way to very quickly test single Python expressions.
...but as soon as you are writing more than a few lines, it will typically be easier to use an editor like VS Code.

The ipython console

IPython

IPython is a more powerful interactive shell

```
$ ipython
Python 3.8.3 (default, May 19 2020, 13:54:14)
Type 'copyright', 'credits' or 'license' for more
information
IPython 7.18.1 -- An enhanced Interactive Python. Type '?'
for help.

In [1]:
```

IPython features

- better support for writing multi-line blocks (like functions)
- syntax coloring
- tab completion
- System shell access: !ls
- magic commands: %timeit, %cd, %pip, ...
- Easy help: ?, ??

IPython feature: multi-line blocks & colors

```
In [1]: def print_str(my_argument):
   ...:     '''Simple printing function'''
   ...:     print(my_argument)
   ...:
In [2]:
```

Unlike in the normal interactive python mode, ipython automatically indents code, and also allows you to go back and edit an entire function at once.

Also note the automatic coloring.

IPython feature: tab completion

```
In [1]: def print_str(my_argument):
   ...:     '''Simple printing function'''
   ...:     print(my_argument)
   ...:
```

Tab completion on names in your namespace:

```
In [2]: pri <PRESS TAB>
              print()     print_str()
```

And on attributes:

```
In [3]: import matplotlib.pyplot as plt

In [4]: plt.pl <PRESS TAB>
                plasma()    plot()
plot_date() plotfile()
```

IPython feature: shell commands

You can execute shell commands directly from the ipython interpreter using the ! prefix.

```
In [1]: !echo "Hello World"
Hello World
```

IPython features: %-magic

IPython has magic commands which can be activated using %.
A few examples:

`%cd` allows you to easily change directory (with tab completion)

```
In [1]: %cd Documents
```

`%timeit` allows you to easily measure time

```
In [2]: %timeit -n 1 -r 1 print("How long does
printing take?")
How long does printing take?
28.5 µs ± 0 ns per loop (mean ± std. dev. of 1 run,
1 loop each)
```

IPython features: help using ? and ??

IPython provides easy access to help using the ? prefix:

```
In [1]: ?range
Init signature: range(self, /, *args, **kwargs)
Docstring:
range(stop) -> range object
range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from
start (inclusive)
to stop (exclusive) by step.  range(i, j) produces i, i+1,
i+2, ..., j-1.
start defaults to 0, and stop is omitted!  range(4)
produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4
elements.
When step is given, it specifies the increment (or
decrement).
Type:           type
Subclasses:
```

Press q to exit

?? provides even more information

IPython features: Input/Output

IPython automatically saves all inputs and outputs

```
In [1]: import numpy as np

In [2]: np.arange(3)
Out[2]: array([0, 1, 2])
```

These inputs and outputs are accessible through the `In` and `Out` variables:

```
In [3]: print(In)
['', 'import numpy as np', 'np.arange(3)', 'print(In)']

In [4]: print(Out)
{2: array([0, 1, 2])}
```

You can manipulate them just as normal values:

```
In [5]: Out[2] * 3
Out[5]: array([0, 3, 6])
```

IPython notebooks

IPython *notebooks* make the IPython functionality available in a web-browser.

Why?

- Interleave code, results and plain text: think of it as an computational lab notebook.
- Inline visualization through matplotlib
- Great for data exploration

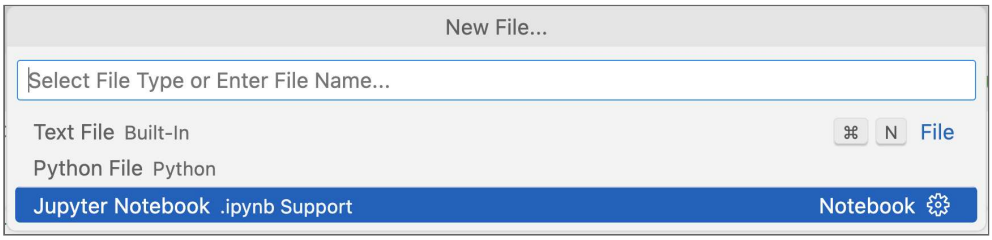
In 2014, the notebook part of IPython was turned into its own project: *Jupyter*, supporting notebooks in multiple languages (the name refers to Julia, Python, R).

These notebooks have become a fundamental tool in Data Science

Jupyter notebooks

Jupyter notebooks

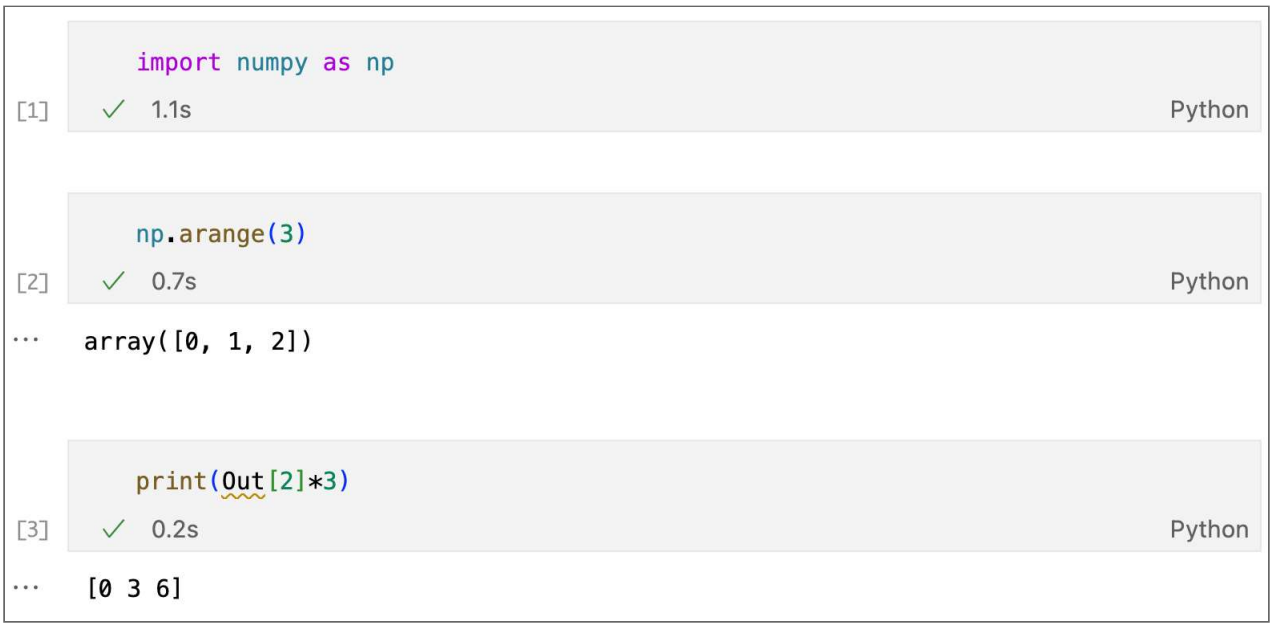
If you have installed the Jupyter extension in VS Code, you can create a new notebook simply by creating a file with the `.ipynb` extension.
...or by choosing Jupyter Notebook in the `File` → `New File` menu



After opening a notebook, make sure to select the correct python version in the top-right corner:

Jupyter notebooks: behaves like IPython

If you press shift-enter after each line of code, Jupyter notebooks will behave like IPython:



Jupyter notebooks

You might see this error when you try to run a line of code:

Run the suggested command in the VS code terminal:

```
$ conda install -n base ipykernel --update-deps --force-
reinstall
```

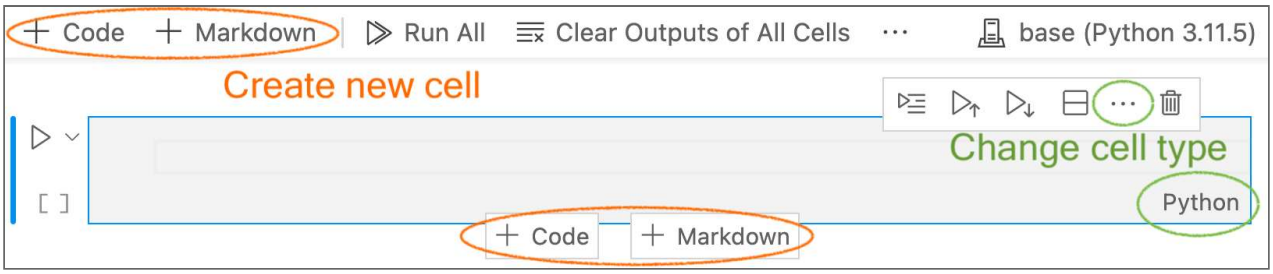
When prompted with `Proceed ([y]/n)?`, type `y+<enter>`

Jupyter notebooks: Cells

Notebooks differ from the IPython console by introducing *cells*.

Different types of cells:

- Code: lines of code
- Markdown: text input using the markdown format



Jupyter notebooks: Markdown cells

Markdown cells allow you to write plain text...
...but you can also add special formatting:

`# header` `## subheader`	Titles of different sizes
`**bold** and *italic*`	Text formatting
`$O(n^2)$` `$$O(n^2)$$`	LaTeX math code (inline or multiline)
`` ```print("hello world!")``` ````	Formatted code
`1. first` `2. second`	Numbered list

And much more. See **the online documentation**.

Jupyter notebooks: Command and edit mode

When working with a notebook, you can be in two modes

- Edit mode
  - Editing the contents of a cell
- Command mode
  - Nagivate through cells, insert cells, ...

Jupyter notebooks: Shortcuts

Both modes

Shift-Enter	Execute cell, select cell below
Alt/Option-Enter	Execute cell, insert new cell below
Ctrl-Enter	Execute selected cells
Ctrl/Command-s	Save

Edit mode

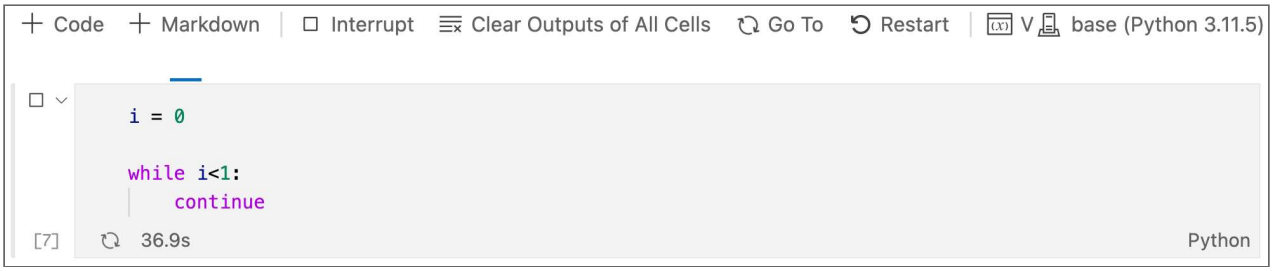
Esc	Switch to Command mode
Tab	Code complete or indent
Enter	Just a normal newline

Command mode

Enter	Switch to Edit mode
Up/Down	Previous/next cell

There are many more, and you can also add your own. See **the online documentation**.

Jupyter notebooks: Restarting the kernel



`Interrupt` (

) is useful for breaking out of long running code executations.

`Restart` (

) is useful for rerunning with a fresh state, so that you are not dependent on invisible old variable values.

`Run all` (

) can be used to execute all cells in order.

Jupyter notebooks: Order matters

```
A = 5
print(A)
```

[1] ✓ 0.2s Python

```
5
```

```
A = A + 2
print(A)
```

[2] ✓ 0.2s Python

```
7
```

```
A = A * 3
print(A)
```

[3] ✓ 0.2s Python

```
21
```

```
A = 5
print(A)
```

[1] ✓ 0.2s Python

```
5
```

```
A = A + 2
print(A)
```

[3] ✓ 0.3s Python

```
17
```

```
A = A * 3
print(A)
```

[2] ✓ 0.2s Python

```
15
```

```
A = 5
print(A)
```

[ ] Python

```
A = A + 2
print(A)
```

[1] ⊗ 0.5s Python

```
---------------------------------------
---------------------------------------
-------
NameError
Traceback (most recent call last)
Cell In[1], line 1
----> 1 A = A + 2
      2 print(A)

NameError: name 'A' is not defined
```

```
A = A * 3
print(A)
```

[ ] Python

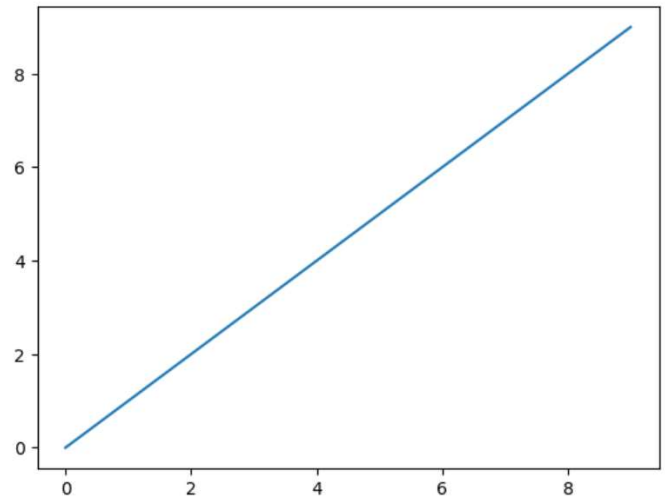
Jupyter notebooks: matplotlib support

Plotting using matplotlib will create inline images

```
[1]  import matplotlib.pyplot as plt
     import numpy as np
     ✓ 0.6s  Python

[2]  x = np.arange(10)
     y = x
     plt.plot(x, y)
     ✓ 0.3s  Python

···  [<matplotlib.lines.Line2D at 0x11aa5ee50>]
```



Notebooks: Pros/Cons

Pros:

- Great for quick prototyping
- Great for exploring data
- Great for educational purposes

Cons:

- Non-linear workflow
- Difficult to do proper version control
- For more reasons see this **entertaining video**

Jupyter notebooks - Exercise

Remember that we discussed simulating trowing two dice in numpy?

```
np.random.randint(1,7,10000) +
np.random.randint(1,7,10000)
```

Let's do this again in a Jupyter notebook:

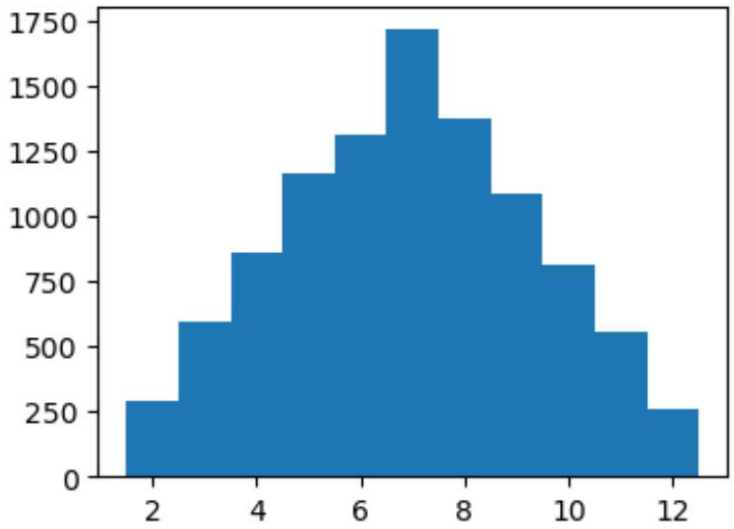
1. Start a new jupyter notebook
2. Execute the line above and save the result
3. Just like we did in the last session, plot a histogram of the results using matplotlib's `plt.hist` function.

Jupyter notebooks - Exercise - solution

```python
import matplotlib.pyplot as plt
import numpy as np

result = np.random.randint(1,7,10000) + np.random.randint(1,7,10000)
plt.hist(result, bins=np.arange(1.5, 13.5, 1))
plt.show()
```

✓ 0.8s Python



Pandas

pandas

What is pandas?

- A library for data manipulation and analysis.
- Make it easy to work with structured tables of numbers
- Much faster than writing loops in Python

In short: whenever you have a list of numbers, consider using numpy and pandas.

Main difference from numpy: data is labeled.

pandas - importing

```
import pandas
```

or (commonly used):

```
import pandas as pd
```

pandas: two data types

Pandas is centered around two data types

- Series
  - Like a 1D numpy array - but with labels
- Dataframe
  - Like a 2D numpy array - but with labels.

pandas - Series

A `Series` is basically a numpy array with an index which defines labels for each entry

```
np_array = np.array([1.0, 2.0, 3.0])

# convert numpy array to pandas series
pd_series = pd.Series(np_array, index=['a', 'b', 'c'])

print(pd_series)
```

```
output

a    1.0
b    2.0
c    3.0
dtype: float64
```

pandas - Series - from a dict

You can also create a Series from a dictionary

```
# Convert dictionary to pandas series
pd_series = pd.Series({'a':1.0, 'b':2.0, 'c':3.0})
print(pd_series)
```

```
output
a    1.0
b    2.0
c    3.0
dtype: float64
```

pandas - Series - behave as arrays and dicts

`Series` behave similar to numpy arrays and dictionaries

```
# Just like numpy arrays, I can operate on all elements at
once
# ... but note how the labels stay aligned
print(pd_series * 2)
```

```
output
a    2.0
b    4.0
c    6.0
dtype: float64
```

```
print(np.log(pd_series))   # I can also use functions from
numpy
```

```
output
a    0.000000
b    0.693147
c    1.098612
dtype: float64
```

```
# Just like dicts, I can ask for a specific key
print(pd_series['b'])
```

```
output
2.0
```

pandas - Series - difference to numpy

Labels are used for alignment

```
print(pd_series + pd_series[1:])
```

```
 output
a    NaN
b    4.0
c    6.0
dtype: float64
```

Very powerful! - no worrying about adding pears to apples
Note that all indices are retained - unless you explicitly drop them:

```
print((pd_series + pd_series[1:]).dropna())
```

```
 output
b    4.0
c    6.0
dtype: float64
```

pandas - accessors

Pandas provides specific support for Series of certain types, through so-called Accessors.

Data type	Accessor
String	str
Categorical	cat
Datetime	dt
Sparse	sparse

For strings, this makes the usual methods available:

```
pd_series = pd.Series({'a':'I',
'b':'love', 'c':'python'})
print(pd_series.str.upper())
```

```
output
a          I
b       LOVE
c     PYTHON
dtype: object
```

This only works for strings. You can convert elements in a series to strings using `.astype('string')`

pandas - Categorical

In addition to the types known from numpy, pandas also has a categorical type
Can be created in different ways:

```
series = pd.Series(['male', 'female', 'female'],
dtype="category")
series = pd.Series(['male', 'female',
'female']).astype('category')
series = pd.Series(pd.Categorical(['male', 'female',
'female']))
series = pd.Series(pd.Categorical(['male', 'female',
'female'],
                                  categories=['female',
'male']))
```

Category specific functionality are accessible through `.cat`:

```
print(series.cat.categories)
```

```
output
Index(['female', 'male'], dtype='object')
```

pandas - Series - Exercise

1. Use range to create a list with values from 0 to 99 and use it to initialize a Series, without providing an index. Print the series and see if you can make sense of the output.
2. Now do the same, but set the index to be from 100 to 199.
3. Convert the entries in the series to type string, and calculate the lengths of each of the entries

pandas - Series - Exercise - solution

1. Use range to create a list with values from 0 to 99 and use it to initialize a Series

```
series = pd.Series(range(100))
print(series)  # A bit confusing, because index and
values are the same
```

2. Now do the same, but let set the range to 5 to 104 instead.

```
series = pd.Series(range(100), index=range(100,200))
print(series)  # Now, we can clearly see the
difference
```

3. Convert the series to type string, and calculate the length of the entries

```
# Convert to string and calculate lengths
series.astype('string').str.len()
```

Pandas: Dataframes

pandas - Dataframe

- The main data type in pandas
- Equivalent to a SAS dataset, an R data frame or an SQL table
- Can be thought of as a `dict` of `Series`

pandas - Dataframe - from numpy array

A dataframe has two sets of labels

- **index:** row labels
- **columns:** column labels

Dataframes can be created from numpy arrays by providing labels for the `index` and the `columns`:

```
np_array = np.arange(6).reshape((2,3))
df = pd.DataFrame(np_array, index=['a', 'b'], columns=
['col1', 'col2', 'col3'])
print(df)
```

```
output
   col1  col2  col3
a     0     1     2
b     3     4     5
```

pandas - Dataframe - from dictionary of lists

You can also initialize from a dictionary of lists

```
dict_of_lists = {'col1': [0,3], 'col2': [1,4], 'col3':
[2,5]}
df = pd.DataFrame(dict_of_lists, index=['a', 'b'])
print(df)
```

```
output
   col1  col2  col3
a     0     1     2
b     3     4     5
```

If `index` is not specified, it will use [0,1, ...]

```
df = pd.DataFrame(dict_of_lists)
print(df)
```

```
output
   col1  col2  col3
0     0     1     2
1     3     4     5
```

pandas - Dataframe - from list of dictionaries

You can also initialize from a lists of dictionaries

```
list_of_dicts = [{'col1': 0, 'col2': 1, 'col3': 2},
                 {'col1': 3, 'col2': 4, 'col3': 5}]
df = pd.DataFrame(list_of_dicts, index=['a', 'b'])
print(df)
```

```
output
   col1  col2  col3
a     0     1     2
b     3     4     5
```

There are many more ways to initialize DataFrames...

pandas - Dataframe: index, columns, values

The labels and values can be accessed using `.index`, `.columns`, and `.values`

```
np_array =
np.arange(6).reshape((3,2))
df = pd.DataFrame(np_array,
                  index=
['a', 'b', 'c'],
                  columns=
['col1', 'col2'])
print(df.index)
print(df.columns)
print(df.values)
```

output

```
Index(['a', 'b', 'c'], dtype='object')
Index(['col1', 'col2'], dtype='object')
[[0 1]
 [2 3]
 [4 5]]
```

pandas - reading/writing in various formats

read_table()	
read_csv()	to_csv()
read_html()	to_html()
read_json()	to_json()
read_hdf()	to_hdf()
read_excel()	to_excel()
read_sas()	
read_sql()	to_sql()
...	

Pandas in Jupyter notebooks

Evaluating a pandas dataframe will render it as HTML

```
[1]  import pandas as pd
     ✓ 4.6s                                                    Python
```

```
[2]  df = pd.DataFrame({'name' : ['Bob', 'Alice', 'Anna'],
                        'sex' : pd.Categorical(['male', 'female', 'female']),
                        'height' : [170, 180, 165]})
     df
     ✓ 0.3s                                                    Python
```

	name	sex	height
0	Bob	male	170
1	Alice	female	180
2	Anna	female	165

Note that when printing a dataframe, it uses the regular layout:

```
[3]  print(df)
     ✓ 0.2s                                                    Python
```

```
    name     sex  height
0    Bob    male     170
1  Alice  female     180
2   Anna  female     165
```

pandas - DataFrame - Exercise

1. Download: **https://wouterboomsma.github.io/ppds2023/data/britis english**
2. Read it into a DataFrame (please use the option `keep_default_na=False` for `read_table`). Are there other options you need to add to make it work?
3. Figure out how to assign a different column name - e.g. 'words'

pandas - DataFrame - Exercise - solution

2. Read it into a DataFrame

```
df = pd.read_table('british-english',
keep_default_na=False, header=None)
```

3. Figure out how to assign a different column name

```
df.columns = ['words']
```

or

```
df.rename(columns={0:'words'}, inplace=True)
```

Speaker notes