Working with Data

Sebastian Ernst, PhD

Course: Data Engineering, EAIiIBISIS.Ii8K.5dfa09851a120.22

Setting up your environment

Jupyter

- **Project Jupyter** is a community developing several **interactive computational** environment tools
- **Jupyter Notebook** is a web-based REPL based on Interactive Python, used to create, view and run **notebook documents**, introduced in 2011
- **JupyterLab** is a newer **user interface**, introduced in 2018
- **JupyterHub** is a **multi-user server** for Jupyter Notebook/JupyterLab instances
- Core supported **languages**: Python, Julia and R

Jupyter Notebook user interface

localhost

jupyter 01-intro Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3 (ipykernel)

Run | Code | git nbdiff

In [30]:
```
d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}
df = pd.DataFrame(d)
df
```

Out[30]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

In [24]:
```
data = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]
pd.DataFrame(data)
```

Out[24]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [28]:
```
del df['b']
```

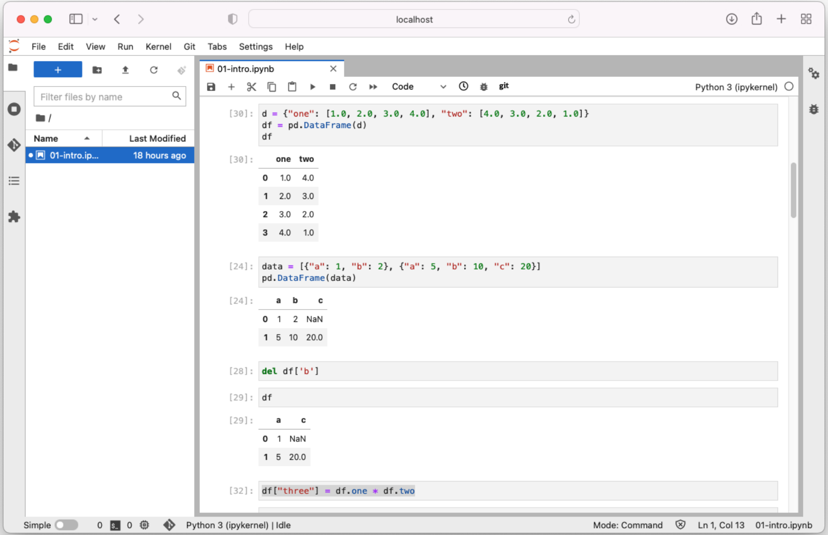
In [29]:
```
df
```

Out[29]:

	a	c
0	1	NaN
1	5	20.0

In [32]:
```
df["three"] = df.one * df.two
```

JupyterLab user interface



Anatomy of a notebook

- A **Jupyter notebook** consists of cells, which can contain:
  - executable **source code**
  - plain **text**
  - formatted **Markdown** text, including **mathematical formulae** (via MathJax)
- **Internally**, a notebook is a **JSON file** with the `.ipynb` extension, containing:
  - cell **contents**,
  - execution **results**,
  - **counters** and **metadata**


```json
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 1,
   "id": "dc7c4ca4-a68a-41ad-90a4-06a13a818dd2",
   "metadata": {},
   "outputs": [],
   "source": [
    "import pandas as pd\n",
    "import numpy as np"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 2,
   "id": "eb21ef4e-f88e-4e66-83a9-49f9b30a6858",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "text/plain": [
       "a    1.151302\n",
       "b   -2.005585\n",
       "c    0.230531\n",
       "d    1.364926\n",
       "e   -0.931755\n",
       "dtype: float64"
      ]
     },
     "execution_count": 2,
     "metadata": {},
     "output_type": "execute_result"
    }
   ],
   "source": [
    "s = pd.Series(np.random.randn(5), index=[\"a\", \"b\", \"c\", \"d\", \"e\"])\n",
    "s"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 3,
   "id": "2434257b-efe4-4fac-8114-0b9aa983a6bc",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "text/plain": [
       "b    1\n",
       "a    0\n",
       "c    2\n",
       "dtype: int64"
      ]
     },
```

Running Jupyter

- **Locally**, by issuing one of the shell commands:

  ```
  jupyter notebook
  jupyter lab
  ```

- These start a **web server** (default **port**: 8888 or first available onwards)
- Authentication using a **token** or a **shared password**
- The **working directory** is the current one, unless specified with the `c.NotebookApp.notebook_dir` parameter
- You can also use **Docker** and one of the available stacks (bind mounts are useful to provide your files)
- Each **opened notebook** gets its own **kernel** – keep a lookout for **neglected kernels** hogging memory

JupyterHub

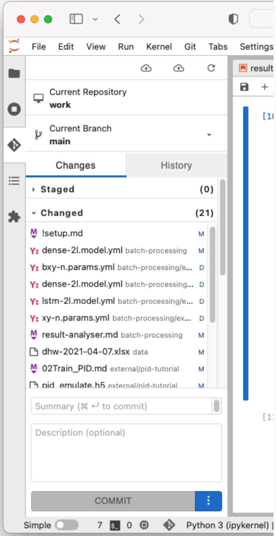
- **Tokens** (or shared passwords) don't really work in **large workgroups**
- Also, **collaborating** on a single notebook is tricky
- JupyterHub provides a **separate instance** of Jupyter Notebook/JupyterLab for **each user**, with a separate **home directory**
- **Used by** many companies and major universities
- Easy to **run** a small instance locally or via Docker; Kubernetes recommended for larger deployments

Jupyter and version control

- As **notebook files** contain counters, results, etc., they **usually don't play nice** with version control
- Solution: maintain **paired versions** of your notebooks via JupyText (and put your `.ipynb` files in `.gitignore`):
  - pairing **formats** include Python scripts and Markdown documents
  - to **pair a notebook**, use a UI command or do it from the command line
  - **automatic pairing** possible by creating a configuration file
- If you wish or need to **version entire notebooks**: nbdime

Added bonus: Git UI in JupyterLab

- **Provided** by jupyterlab-git
- Fully-functional **UI** with support for:
  - staging/commits
  - branches
  - history browsing
  - remotes
- Remember to **set up SSH keys** for remote interaction (JupyterLab's terminal may come in handy)



Running Python on your machine

- Using **system Python** is often a **bad idea**
- Better: use your OS's **package manager**
- Best: use some **Python version management** system

The pip package manager

- The **default package manager** for Python
- Uses the Python Package Index (PyPI) **repository** by default
- Can install **system-wide** (`sudo pip install ...`) or in the **user's home directory** (`pip install --user ...`)
- Also used to install **within environments** (coming up shortly)
- **Freeze** the list of all installed packages: `pip freeze` (but be careful)

Virtual Environments

- The **native way** of creating **separate environments**
- **Creating** an environment:

  ```
  python3 -m venv some-env
  ```

- **Activating** an environment:

  ```
  source some-env/bin/activate
  ```

- **Install packages** using `pip` as usual
- **Downside:** still using the system Python

Conda

- Open-source, cross-platform, language-agnostic **package manager**
- **Distributions:** Anaconda, Miniconda, miniforge (uses conda-forge by default)
- Each **environment** has its own copy Python/pip
- **Create** an environment:

```
conda create -n some-env python=3.9
```

- **Activate** an environment:

```
conda activate some-env
```

- **Install** packages after activating an environment:

```
conda install jupyterlab
```

Switching Python versions with pyenv

- Allows for **per-user** or **per-project** Python version
- Doesn't provide **virtual environments** as such, but has a plugin for it
- Works by inserting a directory of **shims** in the `PATH` environment variable
- Python **version** can be **chosen** using the `PYENV_VERSION` environment variable or through `.python-version` files

```
$ pyenv versions
  2.7.10
* 3.5.0 (set by /Users/yuu/.pyenv/version)
  miniconda3-3.16.0
  pypy-2.6.0

$ python --version
Python 3.5.0

$ pyenv global pypy-2.6.0

$ python --version
Python 2.7.9 (295ee98b69288471b0fcf2e0ede82ce5209eb90b, Jun 01 2015, 17:30:13)
[PyPy 2.6.0 with GCC 4.9.2]

$ cd /Volumes/treasuredata/jupyter

$ pyenv version
miniconda3-3.16.0 (set by /Volumes/treasuredata/.python-version)

$ python --version
Python 3.4.3 :: Continuum Analytics, Inc.
```

Basic data cleaning

Dealing with data types

- When dealing with datasets, we often begin by ensuring the **data types** are **correct** and correctly **interpreted**.
- **Values** can be:
  - of a different **type**, e.g. '*n/a*',
  - **malformed** (e.g. float with a space),
  - **missing**.
- **Check** `df.dtypes`, look out for `object`.

Example CSV file

```
one,two,three,four,five,six,seven
1.0248185874725337,-1.918384453617672,1.4942034191012825,4,red,good,quarrelsome
-0.4257000759609395,-0.004366946015690837,-0.6805007381434629,3,red,bad,doctor
-0.562573262096690l,n/a,-0.027936047483875185,5,red,average,large
-0.34805540449418876,-1.593463352188459,-0.719145576944896 ,5,green,average,muddled
-0.15215280253255076,0.6951082515204178,-0.719145576944896,1,blue,good,coordinated
-0.5680518505468459,1.2004132175276472,-0.719145576944896XYZ,7,blue,good,separate
0.07920837185039101,unknown,-0.24000608193402884,5,green,bad,bright
```

DataFrame .dtypes example

```
>>> df = pd.read_csv('demo.csv')
>>> df
        one                  two                 three  four   five      six        seven
0  1.024819     -1.918384453617672    1.4942034191012825     4    red     good  quarrelsome
1 -0.425700  -0.004366946015690837   -0.6805007381434629     3    red      bad       doctor
2 -0.562573                    NaN  -0.027936047483875185     5    red  average        large
3 -0.348055     -1.593463352188459   -0.719145576944896       5  green  average      muddled
4 -0.152153     0.6951082515204178    -0.719145576944896     1   blue     good  coordinated
5 -0.568052     1.2004132175276472  -0.719145576944896XYZ     7   blue     good     separate
6  0.079208                unknown   -0.24000608193402884     5  green      bad       bright
>>> df.dtypes
one        float64
two         object
three       object
four         int64
five        object
six         object
seven       object
dtype: object
```

Handling string data

- Pandas has extensive **support** for text data, using **string methods** found in the `str` attribute of a `Series`.
- Strings are often filtered using **regular expressions**:

```
>>> s = pd.Series(['joe@here.com', 'jane@there.com',
                   'not.an@email'])
>>> s.str.match(
    r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)")
0     True
1     True
2    False
dtype: bool
```

- If values are **non-atomic**, they can be **split** using Series.str.split() (and optionally expanded).

Handling numeric data

- First, **check the import procedure** for things like decimal commas (instead of points) – see e.g. pd.read_csv options.
- **Convert** strings to a numeric type:
    - the `.astype()` method of DataFrame and Series can **trim whitespace**, but will raise an **exception** or **copy original** values in other cases,
    - pd.to_numeric() is similar, but adds the option to **coerce**,
    - for more elaborate **filtering**, use regular expressions:

```
df.three.str.extract(r'(\-?[0-9]+(?:\.[0-9]+)?)').astype(float)[0]
```

Mapping strings to numbers

- In most cases, **numeric data** is more **useful** than **categorical text data** for analytics or e.g. machine learning.
- If the **categories** are **logically ordered**, map them to adequate numbers:

```
>>> df.six.map({'good': 3, 'average': 2, 'bad': 1})
0    3
1    1
2    2
3    2
4    3
5    3
6    1
Name: six, dtype: int64
```

- The `map` function also accepts `Series` and functions.

One-hot encoding

- If the text labels **do not have an order**, we can use get_dummies() to perform one-hot encoding:

```
>>> pd.get_dummies(df[['five']])
   five_blue  five_green  five_red
0          0           0         1
1          0           0         1
2          0           0         1
3          0           1         0
4          1           0         0
5          1           0         0
6          0           1         0
```

- Note how the **column names** are generated; if the argument was a `Series`, they would only be **label values** (`red`, `green`, `blue`).

Categorical data

- If the values are **non-numeric**, but still belong to a certain **finite set**, it should be represented as the `category` dtype:

```
df = pd.DataFrame({
  "id": [1, 2, 3, 4, 5, 6],
  "raw_grade": ["a", "b", "b", "a", "a", "e"]
})
df["grade"] = df["raw_grade"].astype("category")
```

- Categories can be then easile **renamed**, **reordered**, and **missing categories** can be **added**:

```
df["grade"].cat.categories = ["very good", "good", "very bad"]
df["grade"] = df["grade"].cat.set_categories(
  ["very bad", "bad", "medium", "good", "very good"]
)
```

Dealing with missing data

- Pandas uses `np.na` to **represent** missing values; `isna()` and `notna()` can functions can be used to **detect** them, e.g.:

  ```
  df.isna().sum()
  ```

- Values can be **filled** using `fillna()`, which takes a scaler, dict, `Series` or `Dataframe`.
- We can **drop** rows with missing values using `dropna()`.
- The same can be done with **rows** by changing the *axis*:

  ```
  df.dropna(axis=1)
  ```

Applying lambdas (a.k.a the functional approach)

- Any **function** can be **applied** to a Series or an entire DataFrame
- The function can be a **Python lambda**:

```
df.one.apply(lambda x: 3.14*x)
```

- When applying to a **Series**, the function is called with **scalars** as arguments.
- Applied to a **DataFrame**, the function is called with **Series**:
  - **column-by-column**, argument's index is `df.index`:

    ```
    df.apply(some_function)
    df.apply(some_function, axis=0)
    ```

  - **row-by-row**, argument's index is `df.columns`:

    ```
    df.apply(some_function, axis=1)
    ```

Iterating over pandas objects (a.k.a the procedural approach)

- DataFrames can be **iterated over row-by-row** using `df.iterrows()`:

  ```
  for i, r in df.iterrows():
      print(i)  # index
      print(r)  # row as Series, indexed with df.columns
  ```

- Similarly, we can use `df.items()` to iterate **column-by-column**
- As `df.iterrows()` returns Series, it **does not preserve** dtypes; use `df.itertupes()` if that's needed