

# ***INTRODUCTION TO DATA ANALYSIS AND PLOTTING WITH PANDAS***

## **JSC TUTORIAL**

Andreas Herten, Forschungszentrum Jülich, 26 February 2019

# MY MOTIVATION

- I like Python
- I like plotting data
- I like sharing
- I think Pandas is awesome and you should use it too

*Motto: »Pandas as early as possible!«*

# TASK OUTLINE

- [Task 1](#)
- [Task 2](#)
- [Task 3](#)
- [Task 4](#)
- [Task 5](#)
- [Task 6](#)
- [Task 7](#)
- [Bonus Task](#)

# TUTORIAL SETUP

- 60 minutes (we might do this again for some advanced stuff if you want to)
  - *Well, as it turns out, 60 minutes weren't nearly enough*
  - *We ended up spending nearly 2 hours on it, and we needed to rush quickly through the material*
- Alternating between lecture and hands-on
- Please give status of hands-ons via [pollev.com/aherten538](https://pollev.com/aherten538)
  
- Please open Jupyter Notebook of this session
  - ... either on your **local machine** (`pip install --user pandas seaborn`)
  - ... or on the **JSC Jupyter service** at <https://jupyter-jsc.fz-juelich.de/>  
*Pandas and seaborn should already be there!*
- Tell me when you're done on [pollev.com/aherten538](https://pollev.com/aherten538)

# ABOUT PANDAS

- Python package (Python 2, Python 3)
- For data analysis
- With data structures (multi-dimensional table; time series), operations
- Name from »**Panel Data**« (multi-dimensional time series in economics)
- Since 2008
- <https://pandas.pydata.org/>
- Install via PyPI: `pip install pandas`



# PANDAS COHABITATION

- Pandas works great together with other established Python tools
  - [Jupyter Notebooks](#)
  - Plotting with [matplotlib](#)
  - Modelling with [statsmodels](#), [scikit-learn](#)
  - Nicer plots with [seaborn](#), [altair](#), [plotly](#)

# FIRST STEPS

```
import pandas
```

```
import pandas as pd
```

```
pd.__version__
```

```
'0.24.1'
```

```
%pdoc pd
```

Class docstring:

```
pandas - a powerful data analysis and manipulation library for Python
=====
```

```
**pandas** is a Python package providing fast, flexible, and expressive data
structures designed to make working with "relational" or "labeled" data both
easy and intuitive. It aims to be the fundamental high-level building block for
doing practical, **real world** data analysis in Python. Additionally, it has
the broader goal of becoming **the most powerful and flexible open source data
analysis / manipulation tool available in any language**. It is already well on
its way toward this goal.
```

```
Main Features
```

```
-----
```

```
Here are just a few of the things that pandas does well:
```

- Easy handling of missing data in floating point as well as non-floating

- Size mutability: columns can be inserted and deleted from DataFrame and

# DATAFRAMES

## It's all about DataFrames

- Main data containers of Pandas
  - Linear: `Series`
  - Multi Dimension: `DataFrame`
- `Series` is *only* special case of `DataFrame`
- → Talk about `DataFrames` as the more general case

# DATAFRAMES

## Construction

- To show features of `DataFrame`, let's construct one!
- Many construction possibilities
  - From lists, dictionaries, `numpy` objects
  - From CSV, HDF5, JSON, Excel, HTML, fixed-width files
  - From pickled Pandas data
  - From clipboard
  - *From Feather, Parquest, SAS, SQL, Google BigQuery, STATA*

# DATAFRAMES

## Examples, finally

```
ages = [41, 56, 56, 57, 39, 59, 43, 56, 38, 60]
```

```
pd.DataFrame(ages)
```

0	
0	41
1	56
2	56
3	57
4	39
5	59
6	43
7	56
8	38
9	60

```
df_ages = pd.DataFrame(ages)
df_ages.head(3)
```

0	
0	41
1	56
2	56

- Let's add names to ages; put everything into a `dict()`

```
data = {  
    "Names": ["Liu", "Rowland", "Rivers", "Waters", "Rice", "Fields", "Kerr", "Romero", "Davis", "Hall"],  
    "Ages": ages  
}  
print(data)
```

```
{'Names': ['Liu', 'Rowland', 'Rivers', 'Waters', 'Rice', 'Fields', 'Kerr', 'Romero', 'Davis', 'Hall'], 'Ages':  
[41, 56, 56, 57, 39, 59, 43, 56, 38, 60]}
```

```
df_sample = pd.DataFrame(data)  
df_sample.head(4)
```

	Names	Ages
0	Liu	41
1	Rowland	56
2	Rivers	56
3	Waters	57

- Two columns now; one for names, one for ages

```
df_sample.columns
```

```
Index(['Names', 'Ages'], dtype='object')
```

- DataFrame always have indexes; auto-generated or custom

```
df_sample.index
```

```
RangeIndex(start=0, stop=10, step=1)
```

- Make Names be index with `.set_index()`
- `inplace=True` will modify the parent frame (*I don't like it*)

```
df_sample.set_index("Names", inplace=True)
df_sample
```

Ages	
Names	
Liu	41
Rowland	56
Rivers	56
Waters	57
Rice	39
Fields	59
Kerr	43
Romero	56
Davis	38
Hall	60

- Some more operations

```
df_sample.describe()
```

Ages	
count	10.000000
mean	50.500000
std	9.009255
min	38.000000
25%	41.500000
50%	56.000000
75%	56.750000
max	60.000000

```
df_sample.T
```

Names	Liu	Rowland	Rivers	Waters	Rice	Fields	Kerr	Romero	Davis	Hall
Ages	41	56	56	57	39	59	43	56	38	60

```
df_sample.T.columns
```

```
Index(['Liu', 'Rowland', 'Rivers', 'Waters', 'Rice', 'Fields', 'Kerr',  
      'Romero', 'Davis', 'Hall'],  
      dtype='object', name='Names')
```

- Also: Arithmetic operations

```
df_sample.multiply(2).head(3)
```

Ages	
Names	
Liu	82
Rowland	112
Rivers	112

```
df_sample.reset_index().multiply(2).head(3)
```

	Names	Ages
0	LiuLiu	82
1	RowlandRowland	112
2	RiversRivers	112

```
(df_sample / 2).head(3)
```

Ages	
Names	
Liu	20.5
Rowland	28.0
Rivers	28.0

```
(df_sample * df_sample).head(3)
```

Ages	
Names	
Liu	1681
Rowland	3136
Rivers	3136

Logical operations allowed as well

```
df_sample > 40
```

Ages	
Names	
Liu	True
Rowland	True
Rivers	True
Waters	True
Rice	False
Fields	True
Kerr	True
Romero	True
Davis	False
Hall	True

# TASK 1

- Create data frame with
  - 10 names of dinosaurs,
  - their favourite prime number,
  - and their favourite color
- Play around with the frame
- Tell me on poll when you're done: [pollev.com/aherten538](https://pollev.com/aherten538)

```
happy_dinos = {
  "Dinosaur Name": [],
  "Favourite Prime": [],
  "Favourite Color": []
}
#df_dinos =
```

```
happy_dinos = {
  "Dinosaur Name": ["Aegyptosaurus", "Tyrannosaurus", "Panoplosaurus", "Isisaurus", "Triceratops", "Velociraptor"],
  "Favourite Prime": ["4", "8", "15", "16", "23", "42"],
  "Favourite Color": ["blue", "white", "blue", "purple", "violet", "gray"]
}
df_dinos = pd.DataFrame(happy_dinos).set_index("Dinosaur Name")
df_dinos.T
```

Dinosaur Name	Aegyptosaurus	Tyrannosaurus	Panoplosaurus	Isisaurus	Triceratops	Velociraptor
Favourite Prime	4	8	15	16	23	42
Favourite Color	blue	white	blue	purple	violet	gray

Some more DataFrame examples

```
df_demo = pd.DataFrame({
    "A": 1.2,
    "B": pd.Timestamp('20180226'),
    "C": [(-1)**i * np.sqrt(i) + np.e * (-1)**(i-1) for i in range(5)],
    "D": pd.Categorical(["This", "column", "has", "entries", "entries"]),
    "E": "Same"
})
df_demo
```

	A	B	C	D	E
0	1.2	2018-02-26	-2.718282	This	Same
1	1.2	2018-02-26	1.718282	column	Same
2	1.2	2018-02-26	-1.304068	has	Same
3	1.2	2018-02-26	0.986231	entries	Same
4	1.2	2018-02-26	-0.718282	entries	Same

```
df_demo.sort_values("C")
```

	A	B	C	D	E
0	1.2	2018-02-26	-2.718282	This	Same
2	1.2	2018-02-26	-1.304068	has	Same
4	1.2	2018-02-26	-0.718282	entries	Same
3	1.2	2018-02-26	0.986231	entries	Same
1	1.2	2018-02-26	1.718282	column	Same

```
df_demo.round(2).tail(2)
```

	A	B	C	D	E
3	1.2	2018-02-26	0.99	entries	Same
4	1.2	2018-02-26	-0.72	entries	Same

```
df_demo.round(2).sum()
```

```
A          6
C        -2.03
D  Thiscolumnhasentriesentries
E      SameSameSameSameSame
dtype: object
```

```
print(df_demo.round(2).to_latex())
```

```
\begin{tabular}{lrlrl}
\toprule
{} & A & B & C & D & E \\
\midrule
0 & 1.2 & 2018-02-26 & -2.72 & This & Same \\
1 & 1.2 & 2018-02-26 & 1.72 & column & Same \\
2 & 1.2 & 2018-02-26 & -1.30 & has & Same \\
3 & 1.2 & 2018-02-26 & 0.99 & entries & Same \\
4 & 1.2 & 2018-02-26 & -0.72 & entries & Same \\
\bottomrule
\end{tabular}
```

# READING EXTERNAL DATA

(Links to documentation)

- [`.read\_json\(\)`](#)
- [`.read\_csv\(\)`](#)
- [`.read\_hdf5\(\)`](#)
- [`.read\_excel\(\)`](#)

Example:

```
{
  "Character": ["Sawyer", "...", "Walt"],
  "Actor": ["Josh Holloway", "...", "Malcolm David Kelley"],
  "Main Cast": [true, "...", false]
}
```

```
pd.read_json("lost.json").set_index("Character").sort_index()
```

	Actor	Main Cast
Character		
Hurley	Jorge Garcia	True
Jack	Matthew Fox	True
Kate	Evangeline Lilly	True
Locke	Terry O'Quinn	True
Sawyer	Josh Holloway	True
Walt	Malcolm David Kelley	False

# TASK 2

- Read in `nest-data.csv` to `DataFrame`; call it `df`  
*Data was produced with [JUBE](#), Pandas works **very** well together with [JUBE](#)*
- Get to know it and play a bit with it
- Tell me when you're done: [pollev.com/aherten538](https://pollev.com/aherten538)

```
!cat nest-data.csv | head -3
```

```
id,Nodes,Tasks/Node,Threads/Task,Runtime Program / s,Scale,Plastic,Avg. Neuron Build Time / s,Min. Edge Build Time / s,Max. Edge Build Time / s,Min. Init. Time / s,Max. Init. Time / s,Presim. Time / s,Sim. Time / s,Virt. Memory (Sum) / kB,Local Spike Counter (Sum),Average Rate (Sum),Number of Neurons,Number of Connections,Min. Delay,Max. Delay
5,1,2,4,420.42,10,true,0.29,88.12,88.18,1.14,1.20,17.26,311.52,46560664.00,825499,7.48,112500,1265738500,1.5,1.5
5,1,4,4,200.84,10,true,0.15,46.03,46.34,0.70,1.01,7.87,142.97,46903088.00,802865,7.03,112500,1265738500,1.5,1.5
```

```
df = pd.read_csv("nest-data.csv")
df.head()
```

	id	Nodes	Tasks/Node	Threads/Task	Runtime Program /s	Scale	Plastic	Avg. Neuron Build Time / s	Min. Edge Build Time /s	Max. Edge Build Time /s	...	Max. Init. Time /s	Presim. Time / s	Sim. Time / s	Virt. Memory (Sum) / kB	Local Spike Counter (Sum)	Average Rate (Sum)	Num Neur
0	5	1	2	4	420.42	10	True	0.29	88.12	88.18	...	1.20	17.26	311.52	46560664.0	825499	7.48	1125
1	5	1	4	4	200.84	10	True	0.15	46.03	46.34	...	1.01	7.87	142.97	46903088.0	802865	7.03	1125
2	5	1	2	8	202.15	10	True	0.28	47.98	48.48	...	1.20	7.95	142.81	47699384.0	802865	7.03	1125

# READ CSV OPTIONS

- See also full [API documentation](#)
- Important parameters
  - `sep`: Set separator (for example `:` instead of `,`)
  - `header`: Specify info about headers for columns; able to use multi-index for columns!
  - `names`: Alternative to `header` – provide your own column titles
  - `usecols`: Don't read whole set of columns, but only these; works with any list (`range(0:20:2)`)...
  - `skiprows`: Don't read in these rows
  - `na_values`: What string(s) to recognize as N/A values (which will be ignored during operations on data frame)
  - `parse_dates`: Try to parse dates in CSV; different behaviours as to provided data structure; optionally used together with `date_parser`
  - `compression`: Treat input file as compressed file ("infer", "gzip", "zip", ...)
  - `decimal`: Decimal point divider – for German data...

```
pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.' , lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None)
```

# SLICING OF DATA FRAMES

## Slicing Columns

- Use square-bracket operators to slice data frame: `[]`
  - Use column name to select column
  - Also: Slice horizontally
- Example: Select only columnn `C` from `df_demo`

```
df_demo.head(3)
```

	A	B	C	D	E
0	1.2	2018-02-26	-2.718282	This	Same
1	1.2	2018-02-26	1.718282	column	Same
2	1.2	2018-02-26	-1.304068	has	Same

```
df_demo["C"]
```

```
0    -2.718282
1     1.718282
2    -1.304068
3     0.986231
4    -0.718282
Name: C, dtype: float64
```

- Select more than one column by providing list `[]` to slice operator `[]`
- *You usually end up forgetting one of the brackets...*
- Example: Select list of columns `A` and `C`, `["A", "C"]` from `df_demo`

```
df_demo[["A", "C"]]
```

	A	C
0	1.2	-2.718282
1	1.2	1.718282
2	1.2	-1.304068
3	1.2	0.986231
4	1.2	-0.718282

# SLICING OF DATA FRAMES

## Slicing rows

- Use numerical values to slice into rows
- Use ranges just like with Python lists

```
df_demo[1:3]
```

	A	B	C	D	E
1	1.2	2018-02-26	1.718282	column	Same
2	1.2	2018-02-26	-1.304068	has	Same

- Get a certain range as **per the current sort structure**

```
df_demo.iloc[1:3]
```

	A	B	C	D	E
1	1.2	2018-02-26	1.718282	column	Same
2	1.2	2018-02-26	-1.304068	has	Same

```
df_demo.iloc[1:6:2]
```

	A	B	C	D	E
1	1.2	2018-02-26	1.718282	column	Same
3	1.2	2018-02-26	0.986231	entries	Same

- Attention: `.iloc[]` location might change after re-sorting!

```
df_demo.sort_values("C").iloc[1:3]
```

	A	B	C	D	E
2	1.2	2018-02-26	-1.304068	has	Same
4	1.2	2018-02-26	-0.718282	entries	Same

- One more row-slicing option: `.loc[]`
- See the difference with a *proper* index (and not the auto-generated default index from before)

```
df_demo_indexed = df_demo.set_index("D")
df_demo_indexed
```

	A	B	C	E
D				
This	1.2	2018-02-26	-2.718282	Same
column	1.2	2018-02-26	1.718282	Same
has	1.2	2018-02-26	-1.304068	Same
entries	1.2	2018-02-26	0.986231	Same
entries	1.2	2018-02-26	-0.718282	Same

```
df_demo_indexed.loc["entries"]
```

	A	B	C	E
D				
entries	1.2	2018-02-26	0.986231	Same
entries	1.2	2018-02-26	-0.718282	Same

# ADVANCED SLICING: LOGICAL SLICING

```
df_demo[df_demo["C"] > 0]
```

	A	B	C	D	E
1	1.2	2018-02-26	1.718282	column	Same
3	1.2	2018-02-26	0.986231	entries	Same

```
df_demo[(df_demo["C"] < 0) & (df_demo["D"] == "entries")]
```

	A	B	C	D	E
4	1.2	2018-02-26	-0.718282	entries	Same

# ADDING TO EXISTING DATA FRAME

- Add new columns with `frame["new col"] = something` or `.insert()`
- Add new rows with `frame.append()`
- Combine data frames
  - *Concat*: Combine several data frames along an axis
  - *Merge*: Combine data frames on basis of common columns; database-style
  - (Join)
  - See user guide [on merging](#)

```
df_demo.head(3)
```

	A	B	C	D	E
0	1.2	2018-02-26	-2.718282	This	Same
1	1.2	2018-02-26	1.718282	column	Same
2	1.2	2018-02-26	-1.304068	has	Same

```
df_demo["F"] = df_demo["C"] - df_demo["A"]
df_demo.head(3)
```

	A	B	C	D	E	F
0	1.2	2018-02-26	-2.718282	This	Same	-3.918282
1	1.2	2018-02-26	1.718282	column	Same	0.518282
2	1.2	2018-02-26	-1.304068	has	Same	-2.504068

```
df_demo.insert(df_demo.shape[1], "G", df_demo["C"] ** 2)
```

```
df_demo.tail(3)
```

	A	B	C	D	E	F	G
2	1.2	2018-02-26	-1.304068	has	Same	-2.504068	1.700594
3	1.2	2018-02-26	0.986231	entries	Same	-0.213769	0.972652
4	1.2	2018-02-26	-0.718282	entries	Same	-1.918282	0.515929

```
df_demo.append(  
    {"A": 1.3, "B": pd.Timestamp("2018-02-27"), "C": -0.777, "D": "has it?", "E": "Same", "F": 23},  
    ignore_index=True  
)
```

	A	B	C	D	E	F	G
0	1.2	2018-02-26	-2.718282	This	Same	-3.918282	7.389056
1	1.2	2018-02-26	1.718282	column	Same	0.518282	2.952492
2	1.2	2018-02-26	-1.304068	has	Same	-2.504068	1.700594
3	1.2	2018-02-26	0.986231	entries	Same	-0.213769	0.972652
4	1.2	2018-02-26	-0.718282	entries	Same	-1.918282	0.515929
5	1.3	2018-02-27	-0.777000	has it?	Same	23.000000	NaN

# COMBINING FRAMES

- First, create some simpler data frame to show `.concat()` and `.merge()`

```
df_1 = pd.DataFrame({"Key": ["First", "Second"], "Value": [1, 1]})
df_1
```

	Key	Value
0	First	1
1	Second	1

```
df_2 = pd.DataFrame({"Key": ["First", "Second"], "Value": [2, 2]})
df_2
```

	Key	Value
0	First	2
1	Second	2

- Concatenate list of data frame vertically (axis=0)

```
pd.concat([df_1, df_2])
```

	Key	Value
0	First	1
1	Second	1
0	First	2
1	Second	2

- Same, but re-index

```
pd.concat([df_1, df_2], ignore_index=True)
```

	Key	Value
0	First	1
1	Second	1
2	First	2
3	Second	2

- Concat, but horizontally

```
pd.concat([df_1, df_2], axis=1)
```

	Key	Value	Key	Value
0	First	1	First	2
1	Second	1	Second	2

- Merge on common column

```
pd.merge(df_1, df_2, on="Key")
```

	Key	Value_x	Value_y
0	First	1	2
1	Second	1	2

TASK 3

- Add a column to the Nest data frame called Virtual Processes which is the total number of threads across all nodes (i.e. the product of threads per task and tasks per node and nodes)
- Remember to tell me when you're done: [pollev.com/aherten538](https://pollev.com/aherten538)

```
df["Virtual Processes"] = df["Nodes"] * df["Tasks/Node"] * df["Threads/Task"]
df.head()
```

	id	Nodes	Tasks/Node	Threads/Task	Runtime Program /s	Scale	Plastic	Avg. Neuron Build Time / s	Min. Edge Build Time /s	Max. Edge Build Time /s	...	Presim. Time / s	Sim. Time / s	Virt. Memory (Sum) / kB	Local Spike Counter (Sum)	Average Rate (Sum)	Number of Neurons	C
0	5	1	2	4	420.42	10	True	0.29	88.12	88.18	...	17.26	311.52	46560664.0	825499	7.48	112500	1
1	5	1	4	4	200.84	10	True	0.15	46.03	46.34	...	7.87	142.97	46903088.0	802865	7.03	112500	1
2	5	1	2	8	202.15	10	True	0.28	47.98	48.48	...	7.95	142.81	47699384.0	802865	7.03	112500	1
3	5	1	4	8	89.57	10	True	0.15	20.41	23.21	...	3.19	60.31	46813040.0	821491	7.23	112500	1
4	5	2	2	4	164.16	10	True	0.20	40.03	41.09	...	6.08	114.88	46937216.0	802865	7.03	112500	1

5 rows × 22 columns

```
df.columns
```

```
Index(['id', 'Nodes', 'Tasks/Node', 'Threads/Task', 'Runtime Program / s',  
      'Scale', 'Plastic', 'Avg. Neuron Build Time / s',  
      'Min. Edge Build Time / s', 'Max. Edge Build Time / s',  
      'Min. Init. Time / s', 'Max. Init. Time / s', 'Presim. Time / s',
```

# ASIDE: PLOTTING WITHOUT PANDAS

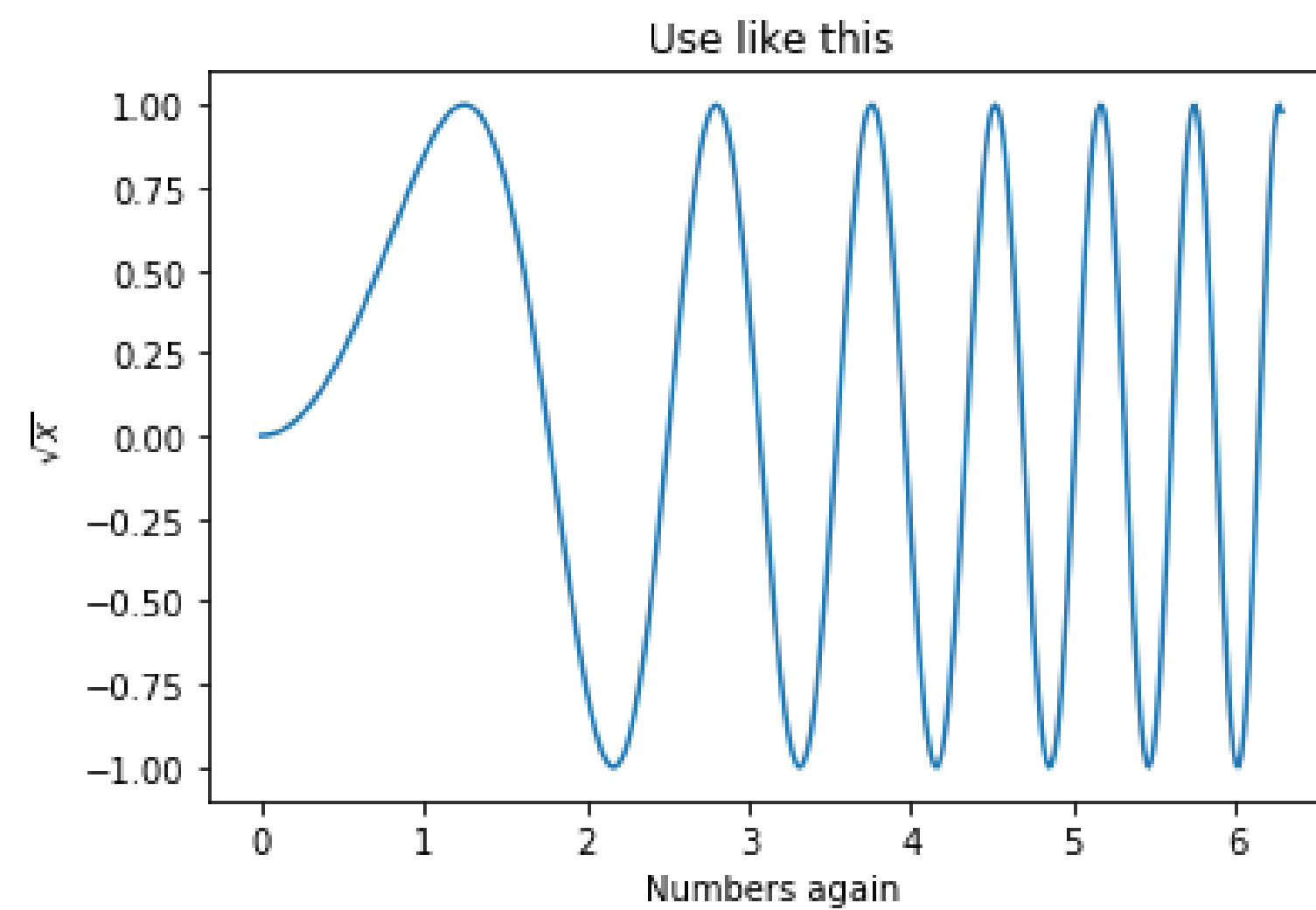
## Matplotlib 101

- Matplotlib: de-facto standard for plotting in Python
- Main interface: `pyplot`; provides MATLAB-like interface
- Better: Use object-oriented API with `Figure` and `Axis`
- Great integration into Jupyter Notebooks
- Since v. 3: Only support for Python 3
- → <https://matplotlib.org/>

```
import matplotlib.pyplot as plt
%matplotlib inline
```

```
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)
```

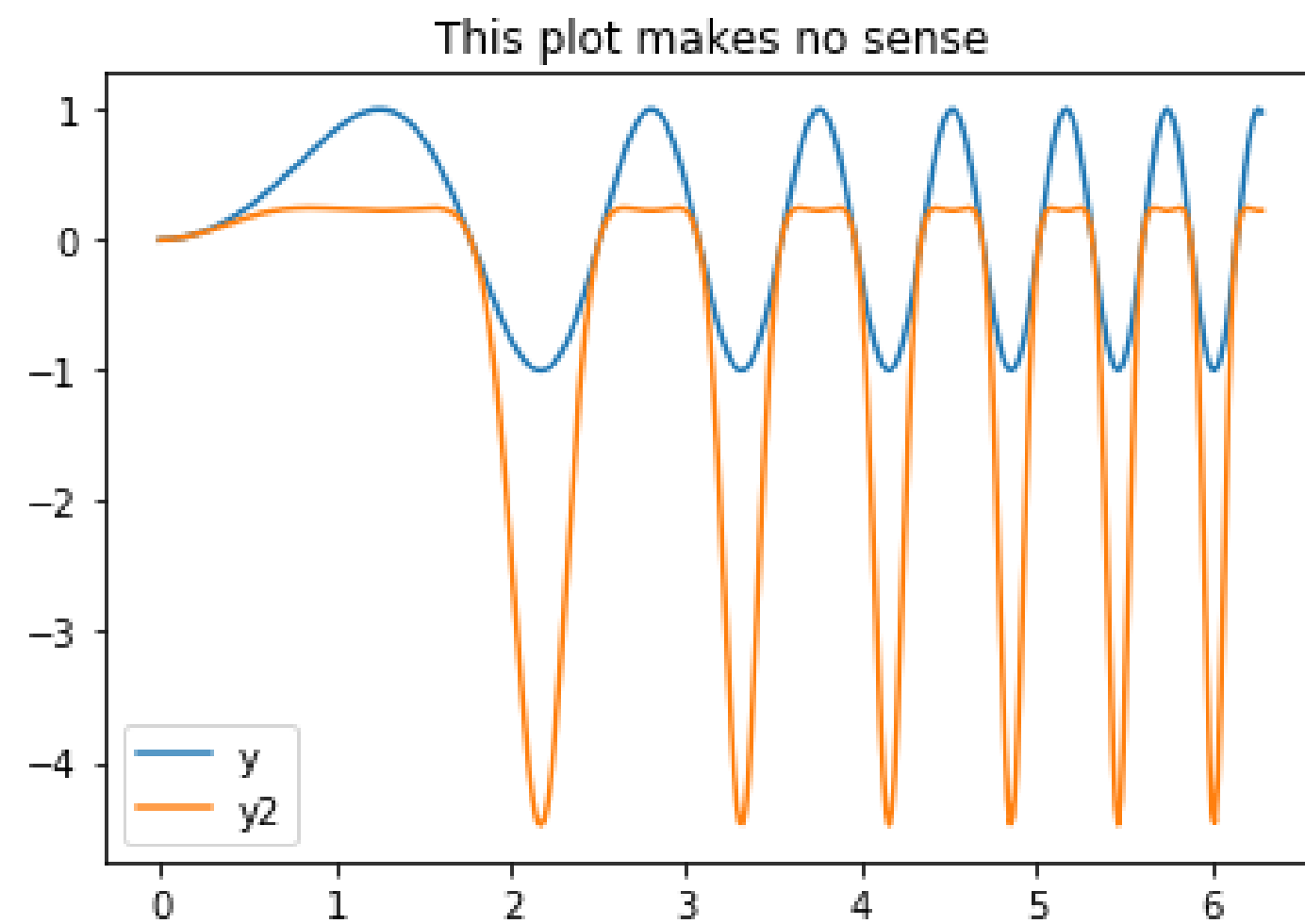
```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Use like this')
ax.set_xlabel("Numbers again");
ax.set_ylabel("$\sqrt{x}$");
```



- Plot multiple lines into one canvas
- Call `ax.plot()` multiple times

```
y2 = y/np.exp(y*1.5)
```

```
fig, ax = plt.subplots()
ax.plot(x, y, label="y")
ax.plot(x, y2, label="y2")
ax.legend()
ax.set_title("This plot makes no sense");
```



## TASK 4

- Sort the data frame by the virtual proccesses
- Plot "Presim. Time / s" and "Sim. Time / s" of our data frame `df` as a function of the virtual processes
- Use a dashed, red line for "Presim. Time / s", a blue line for "Sim. Time / s" (see [API description](#))
- Don't forget to label your axes and to add a legend
- Submit when you're done: [pollev.com/aherten538](https://pollev.com/aherten538)

```
df.sort_values(["Virtual Processes", "Nodes", "Tasks/Node", "Threads/Task"], inplace=True)
```

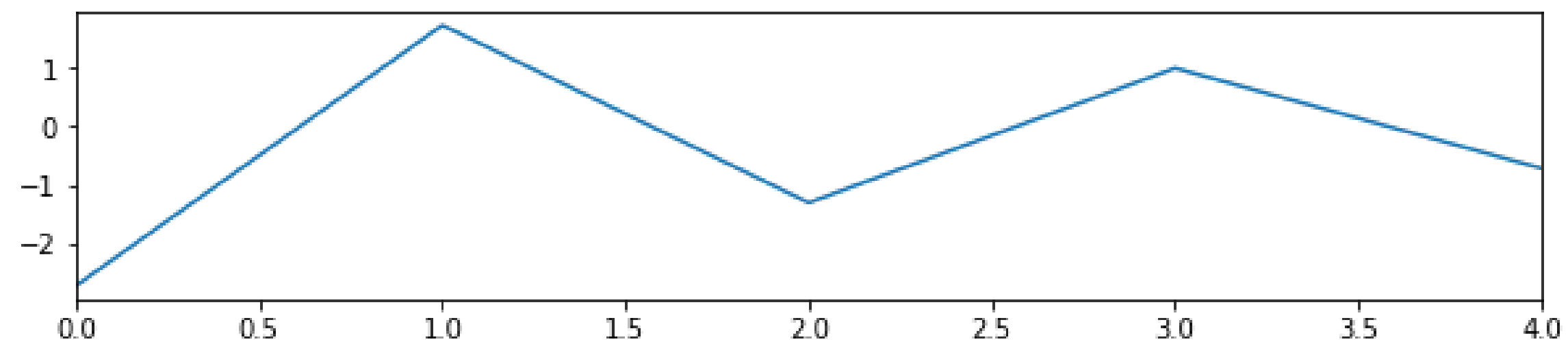
```
fig, ax = plt.subplots()
ax.plot(df["Virtual Processes"], df["Presim. Time / s"], linestyle="dashed", color="red")
ax.plot(df["Virtual Processes"], df["Sim. Time / s"], "-b")
ax.set_xlabel("Virtual Process")
ax.set_ylabel("Time / s")
ax.legend();
```

# PLOTTING WITH PANDAS

- Each data frame has a `.plot()` function (see [API](#))
- Plots with Matplotlib
- Important API options:
  - `kind`: `line` (default), `bar`, `h`, `hist`, `box`, `kde`, `scatter`, `hexbin`
  - `subplots`: Make a sub-plot for each column (good together with `sharex`, `sharey`)
  - `figsize`
  - `grid`: Add a grid to plot (use Matplotlib options)
  - `style`: Line style per column (accepts list or dict)
  - `logx`, `logy`, `loglog`: Logarithmic plots
  - `xticks`, `yticks`: Use values for ticks
  - `xlim`, `ylim`: Limits of axes
  - `yerr`, `xerr`: Add uncertainty to data points
  - `stacked`: Stack a bar plot
  - `secondary_y`: Use a secondary `y` axis for this plot
  - Labeling
    - `title`: Add title to plot (Use a list of strings if `subplots=True`)
    - `legend`: Add a legend
    - `table`: If `true`, add table of data under plot
  - `**kwargs`: Every non-parsed keyword is passed through to Matplotlib's plotting methods

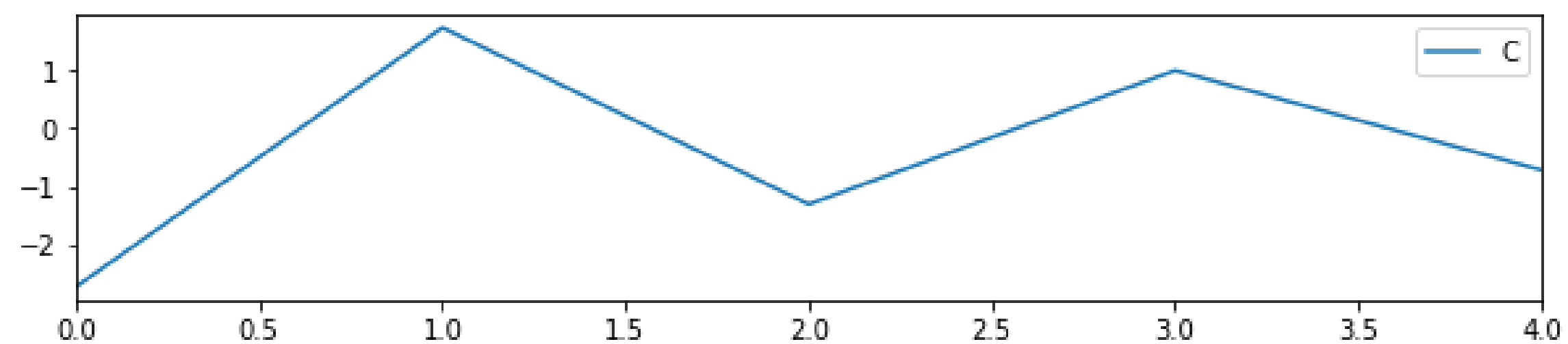
- Either slice and plot...

```
df_demo["C"].plot(figsize=(10, 2));
```



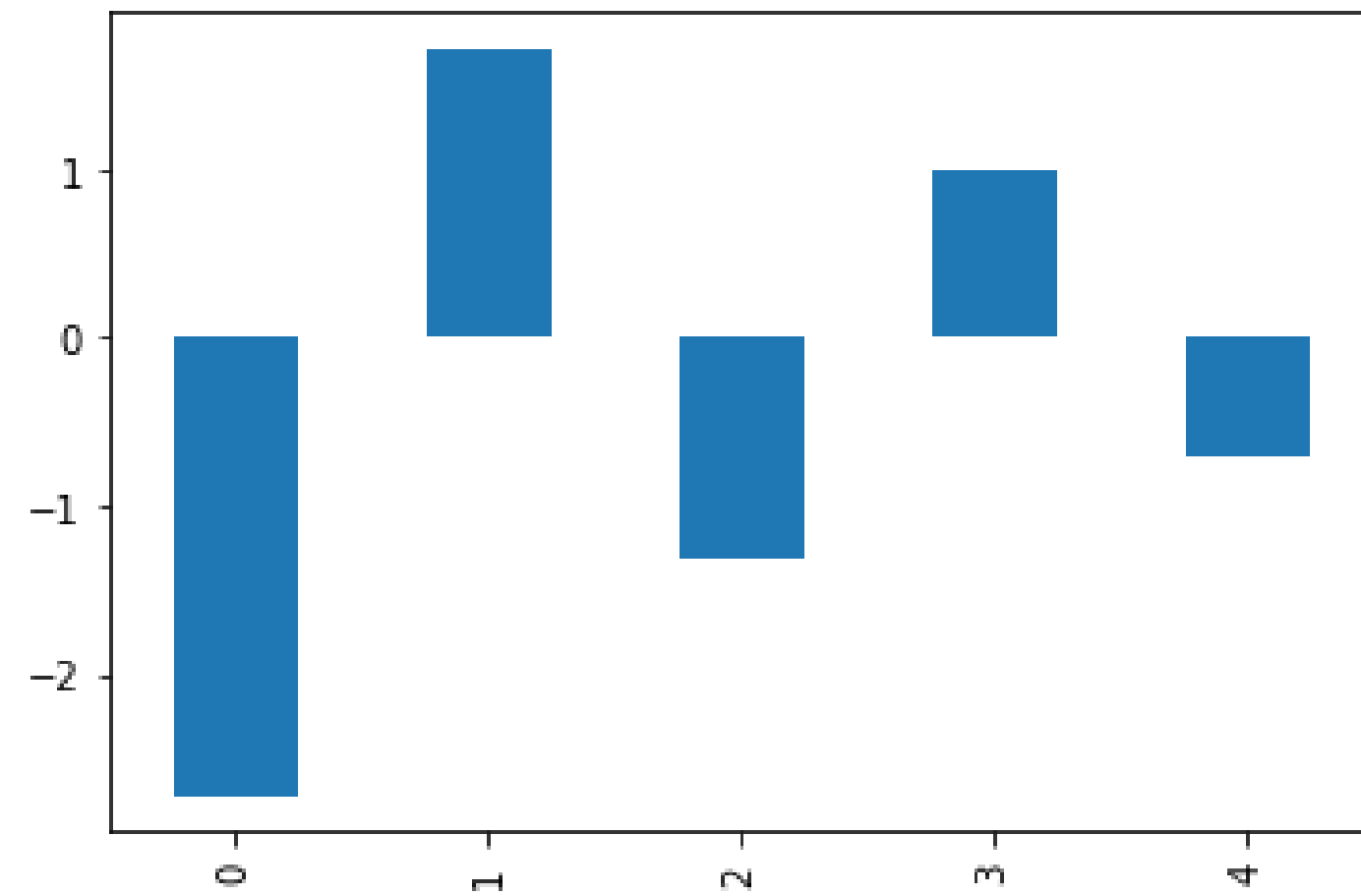
- ... or plot and select

```
df_demo.plot(y="C", figsize=(10, 2));
```



- I prefer slicing first, as it allows for further operations on the sliced data frame

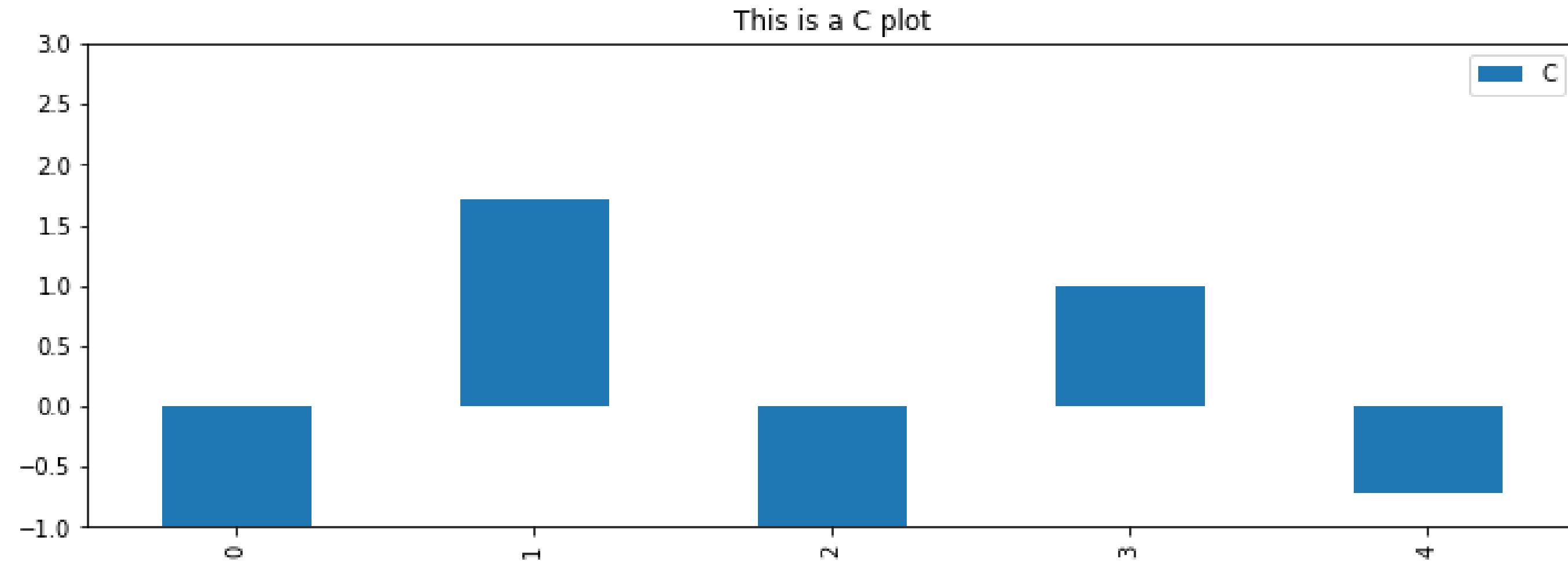
```
df_demo["C"].plot(kind="bar");
```



- There are pseudo-sub-functions for each of the plot kinds
- I prefer to just call `.plot(kind="smthng")`

```
df_demo["C"].plot.bar();
```

```
df_demo["C"].plot(kind="bar", legend=True, figsize=(12, 4), ylim=(-1, 3), title="This is a C plot");
```



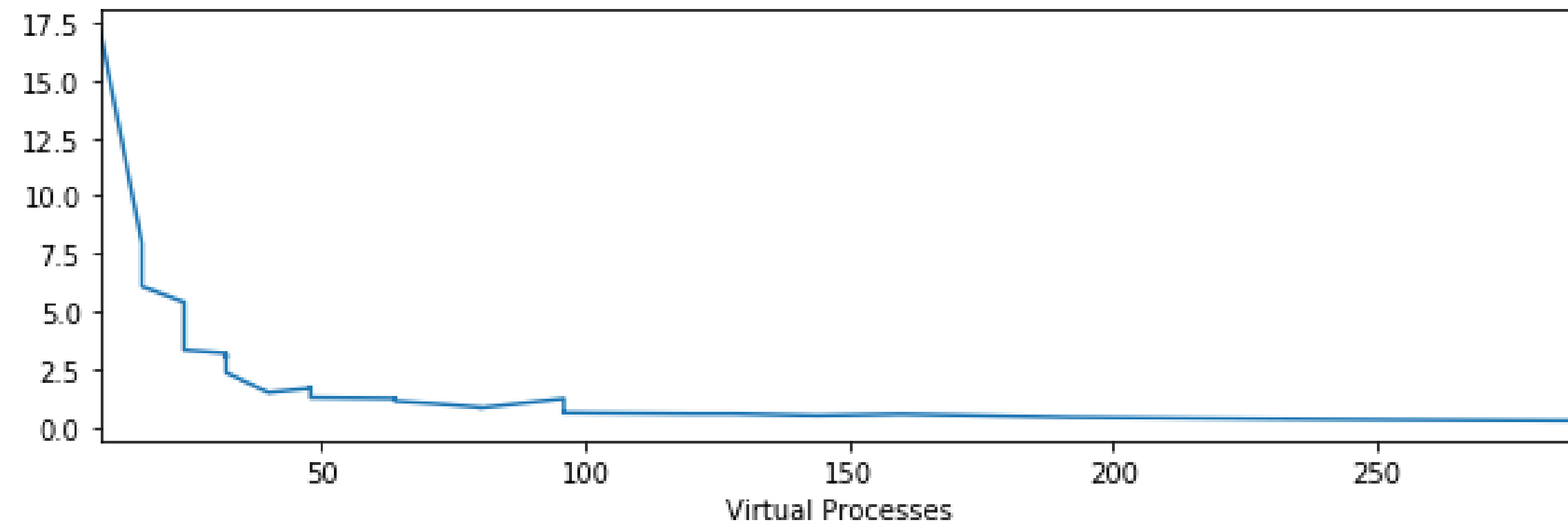
## TASK 5

Use the NEST data frame `df` to:

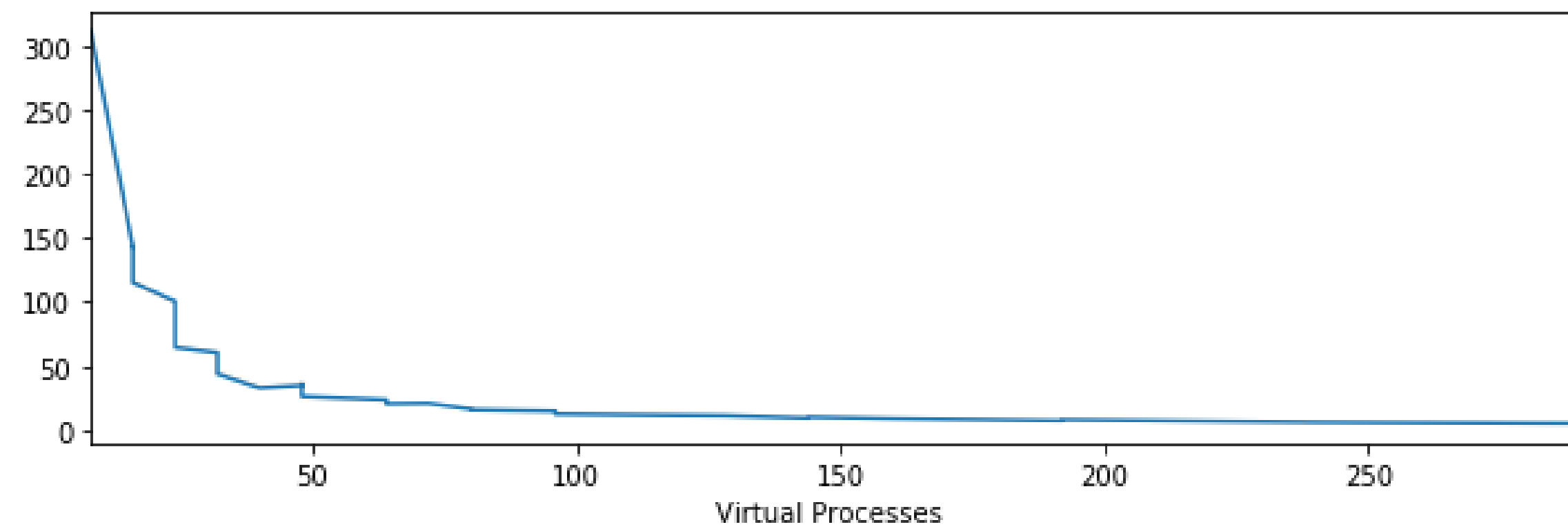
1. Make the virtual processes the index of the data frame (`.set_index()`)
2. Plot `"Presim. Program / s"` and `"Sim. Time / s"` individually
3. Plot them onto one common canvas!
4. Make them have the same line colors and styles as before
5. Add a legend, add missing labels
6. Done? Tell me! [pollev.com/aherten538](https://pollev.com/aherten538)

```
df.set_index("Virtual Processes", inplace=True)
```

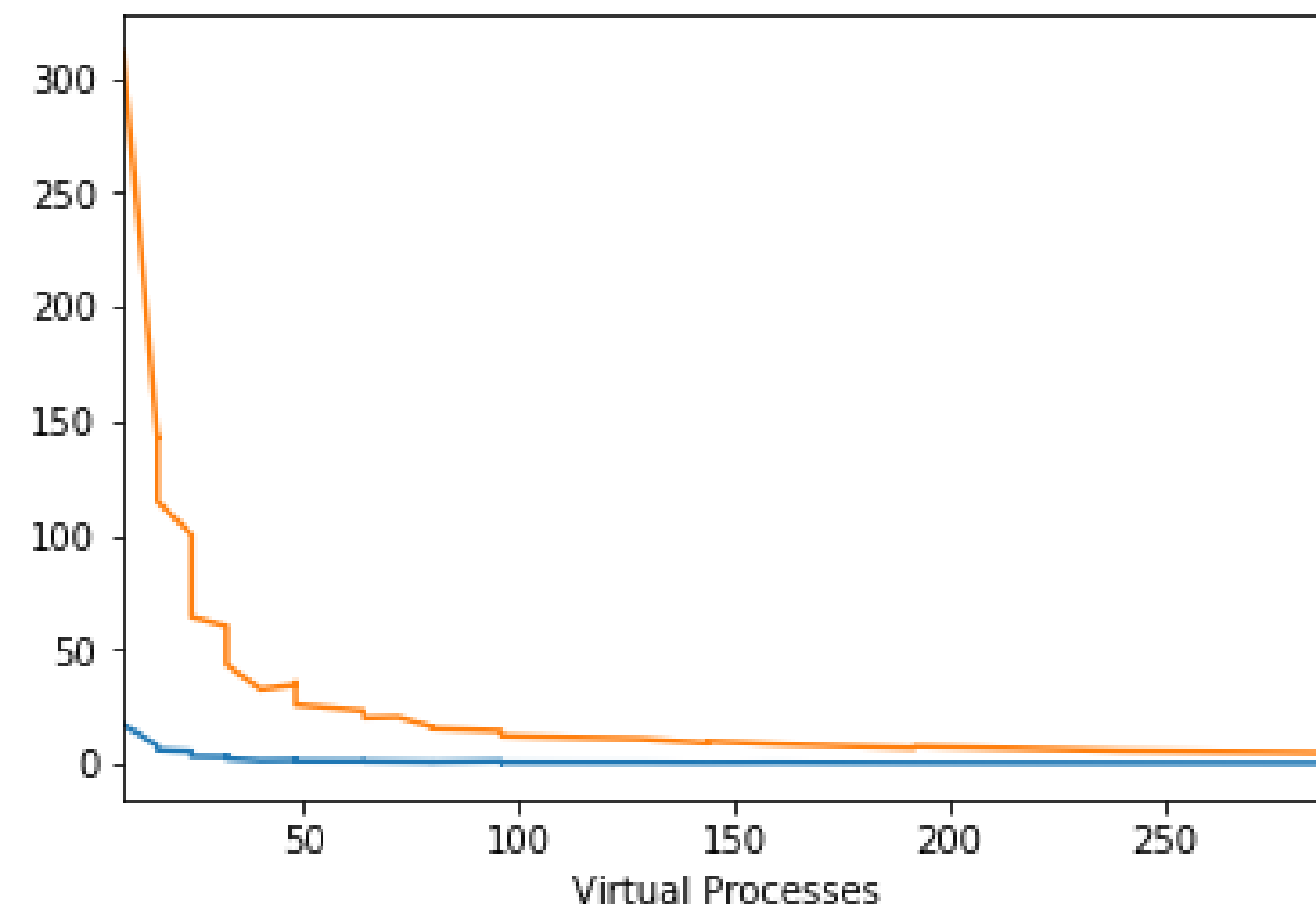
```
df["Presim. Time / s"].plot(figsize=(10, 3));
```



```
df["Sim. Time / s"].plot(figsize=(10, 3));
```



```
df["Presim. Time / s"].plot();  
df["Sim. Time / s"].plot();
```

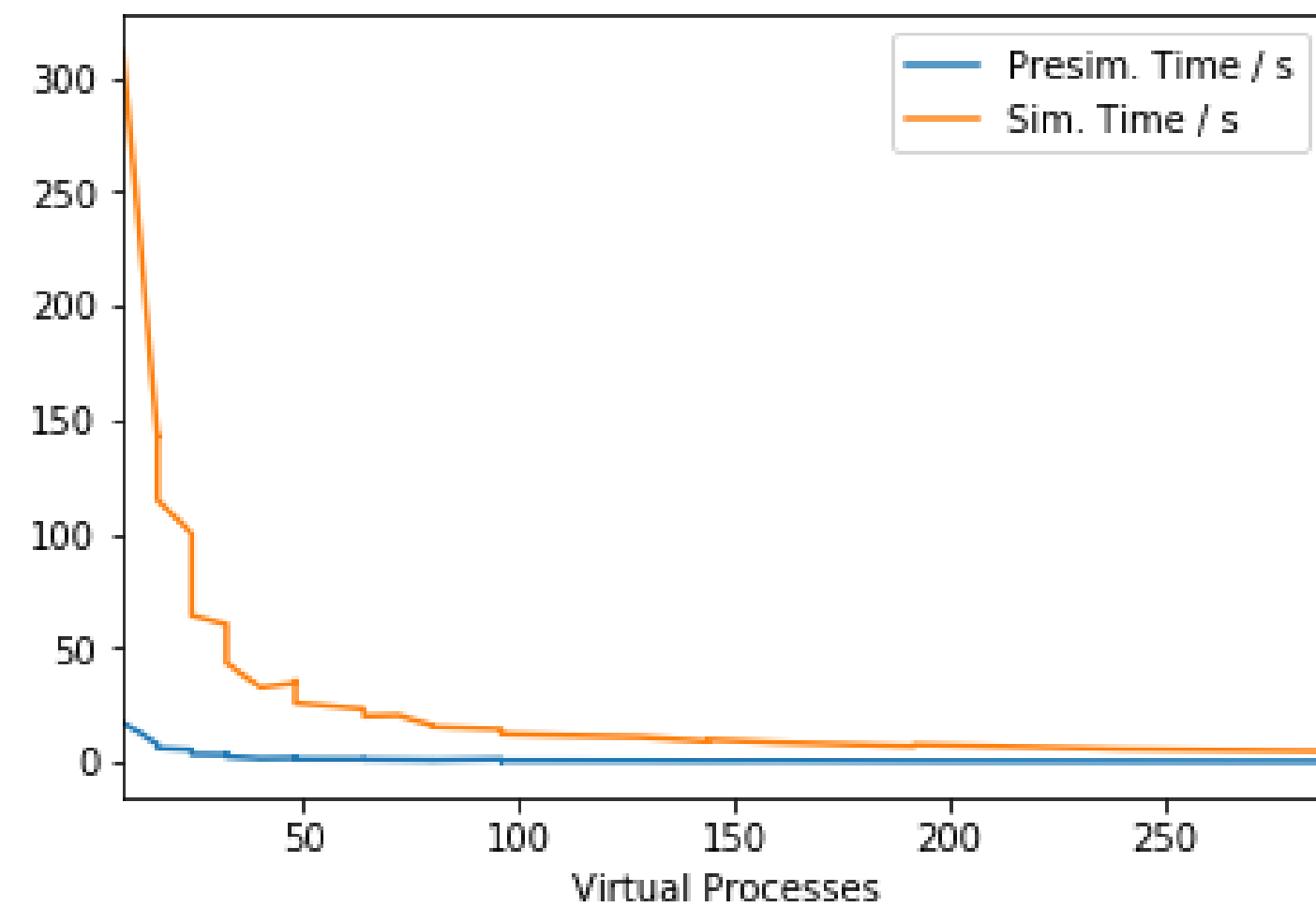


```
ax = df[["Presim. Time / s", "Sim. Time / s"]].plot();  
ax.set_ylabel("Time / s");
```

# MORE PLOTTING WITH PANDAS

## Our first proper Pandas plot

```
df[["Presim. Time / s", "Sim. Time / s"]].plot();
```

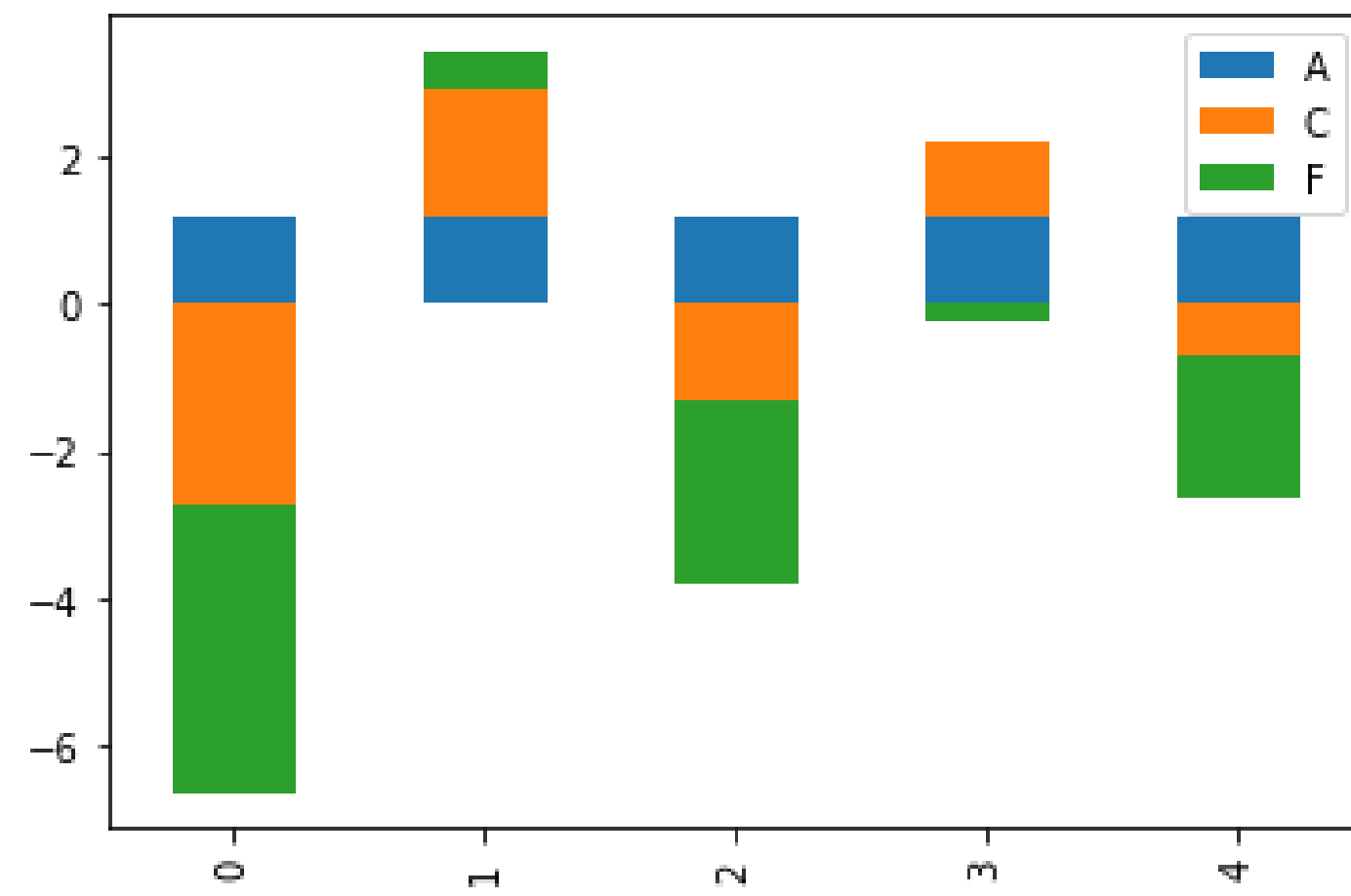


- That's why I think Pandas is great!
- It has great defaults to quickly plot data
- Plotting functionality is very versatile
- Before plotting, data can be *massaged* within data frames, if needed

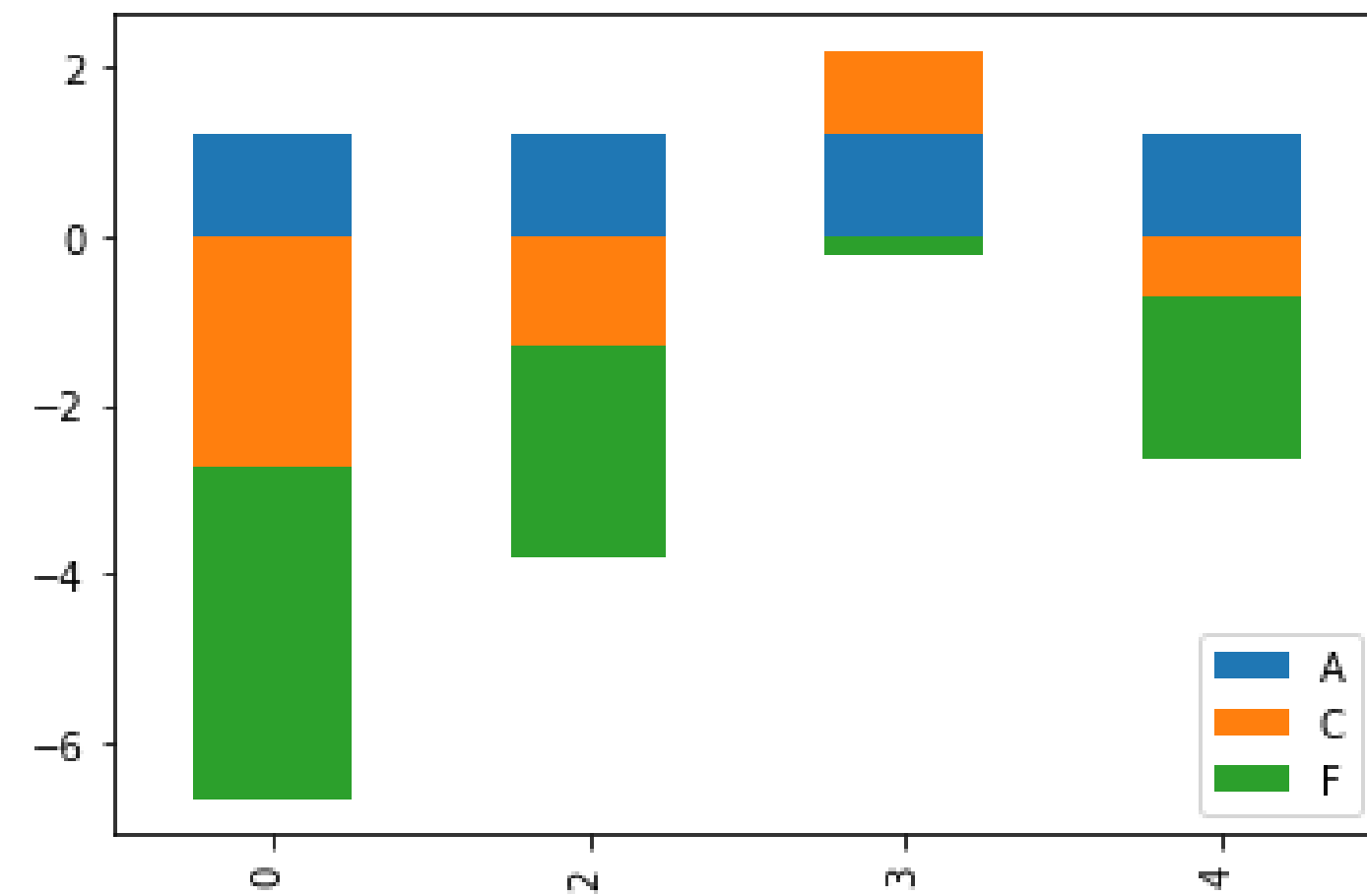
# MORE PLOTTING WITH PANDAS

## Some versatility

```
df_demo[["A", "C", "F"]].plot(kind="bar", stacked=True);
```

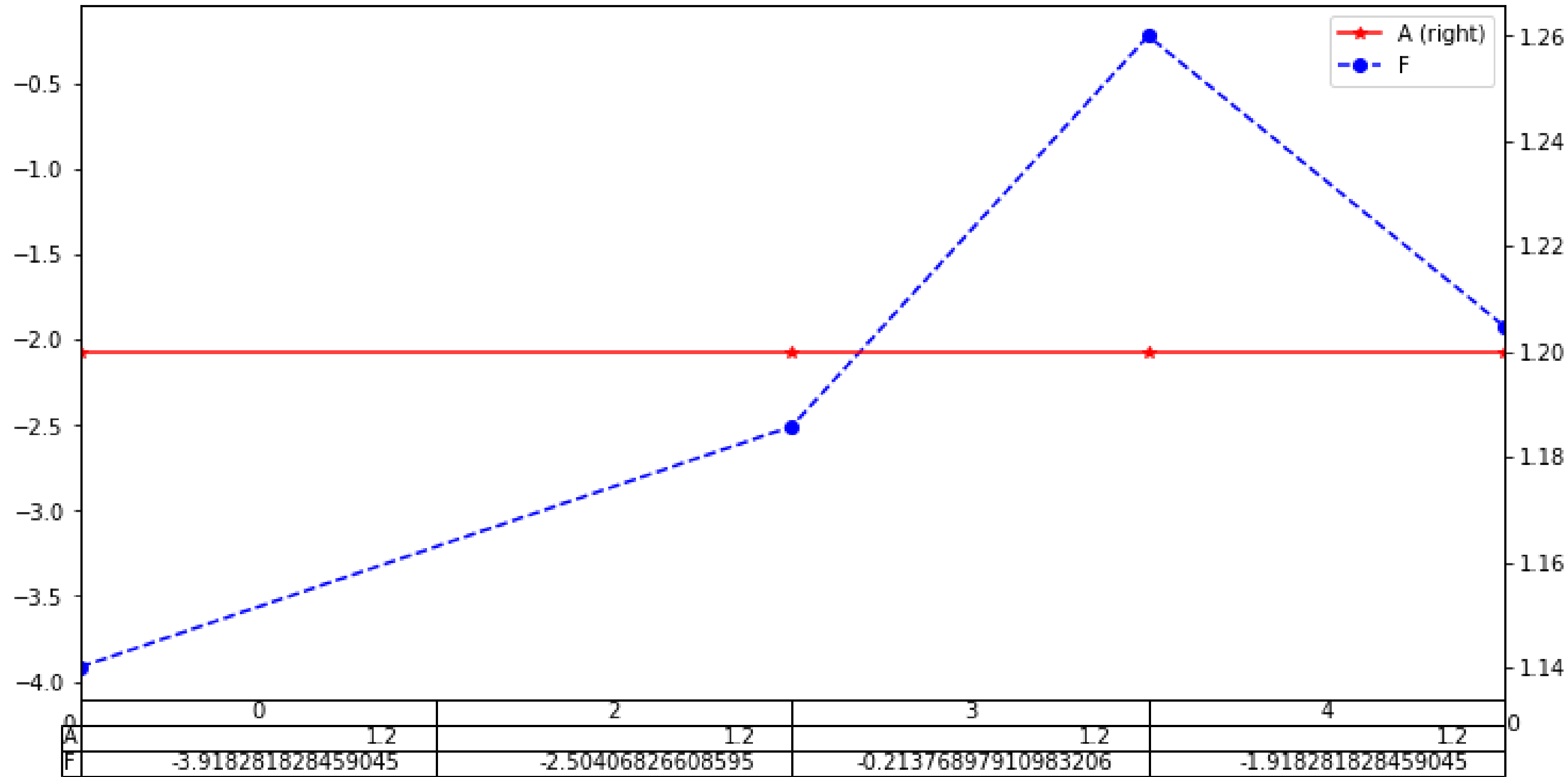


```
df_demo[df_demo["F"] < 0][["A", "C", "F"]].plot(kind="bar", stacked=True);
```

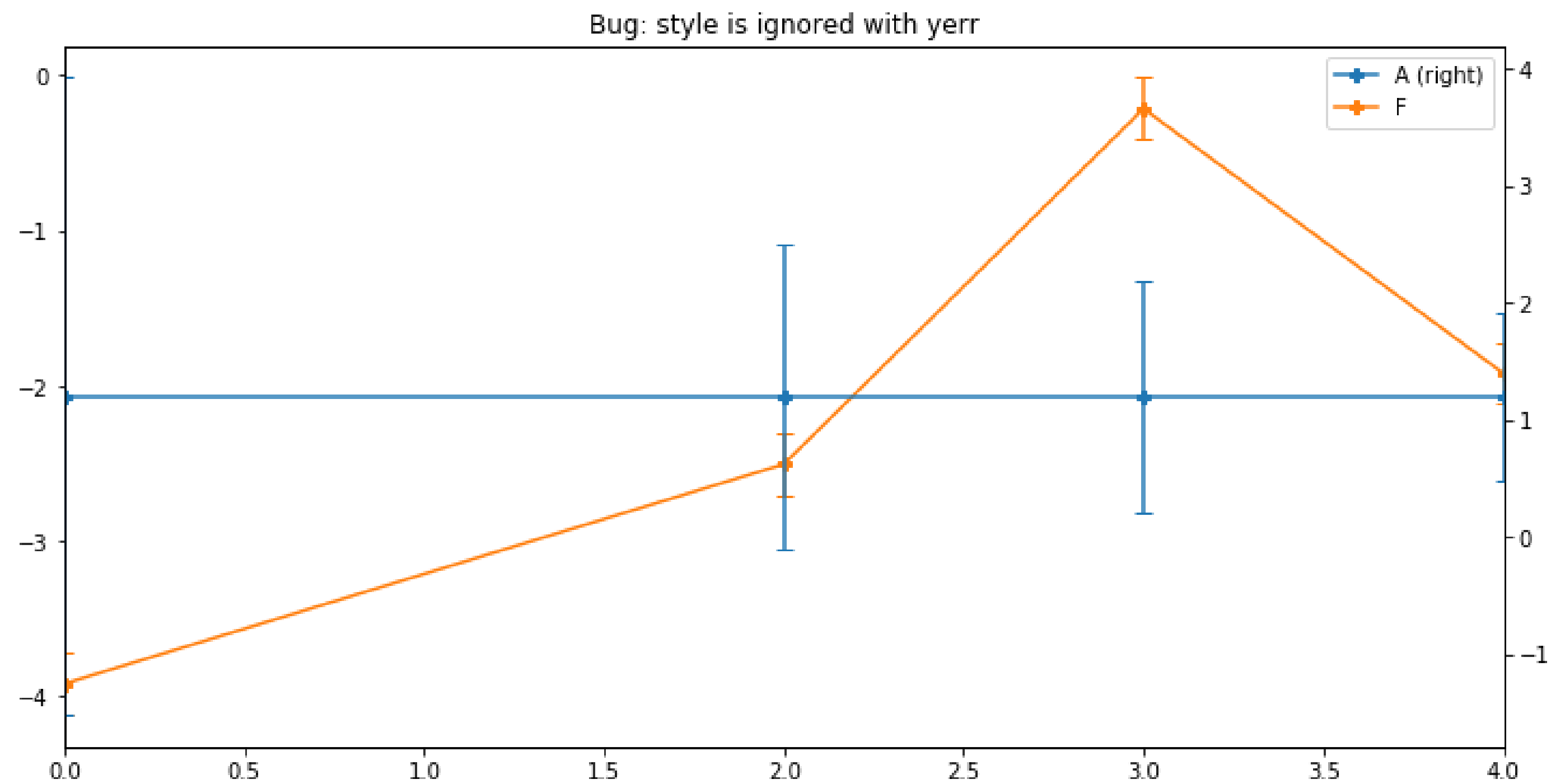


```
df_demo[df_demo["F"] < 0][["A", "C", "F"]]\n    .plot(kind="barh", subplots=True, sharex=True, title="Subplots", figsize=(12, 4));
```

```
df_demo[df_demo["F"] < 0][["A", "F"]]\n    .plot(\n        style=["-*r", "--ob"],\n        secondary_y="A",\n        figsize=(12, 6),\n        table=True\n    );
```



```
df_demo[df_demo["F"] < 0][["A", "F"]]\
    .plot(
        style=["-*r", "--ob"],
        secondary_y="A",
        figsize=(12, 6),
        yerr={
            "A": df_demo[df_demo["F"] < 0]["C"],
            "F": 0.2
        },
        capsize=4,
        title="Bug: style is ignored with yerr",
        marker="P"
    );
```

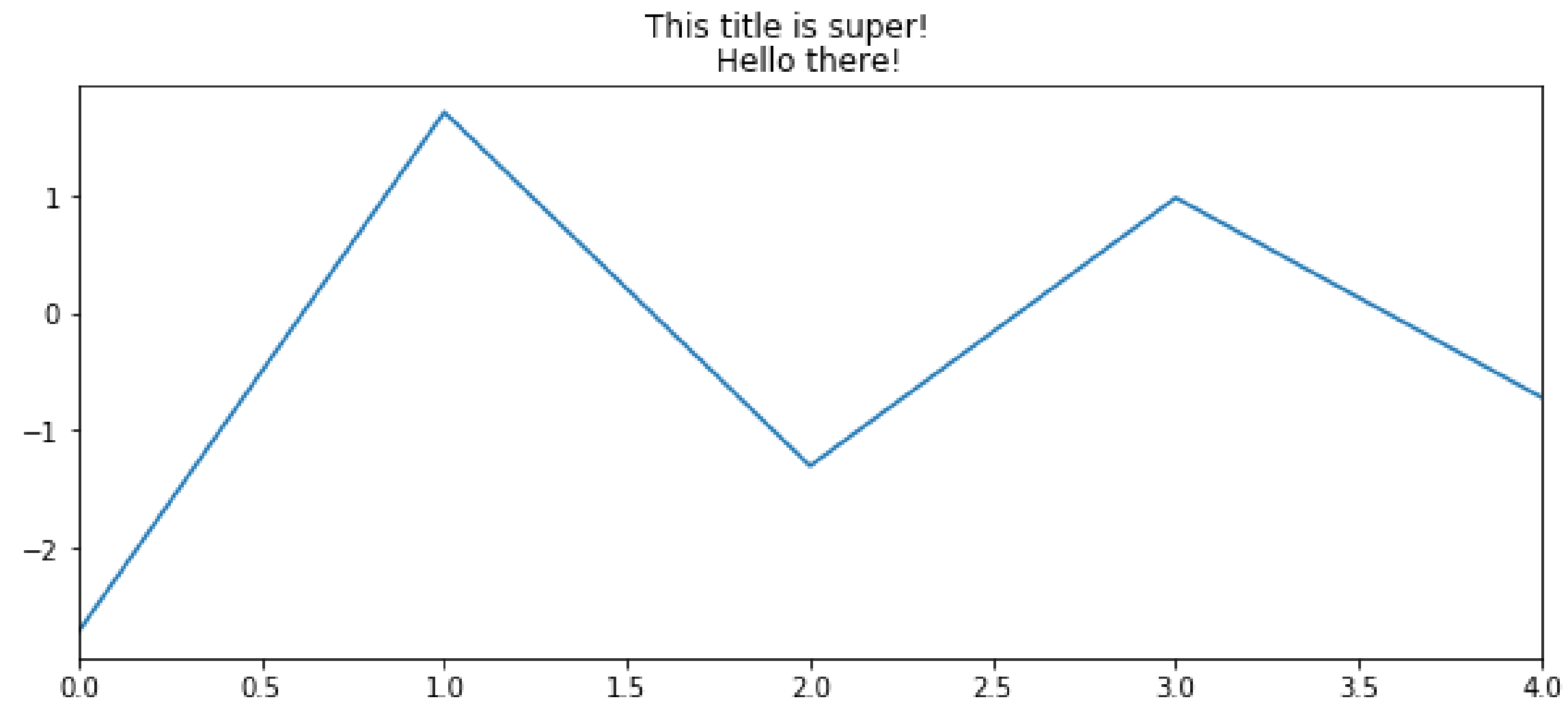


# COMBINE PANDAS WITH MATPLOTLIB

- Pandas shortcuts very handy
- But sometimes, one needs to access underlying Matplotlib functionality
- No problemo!
- **Option 1:** Pandas always returns axis
  - Use this to manipulate the canvas
  - Get underlying `figure` with `ax.get_figure()` (for `fig.savefig()`)
- **Option 2:** Create figure and axes with Matplotlib, use when drawing
  - `.plot()`: Use `ax` option

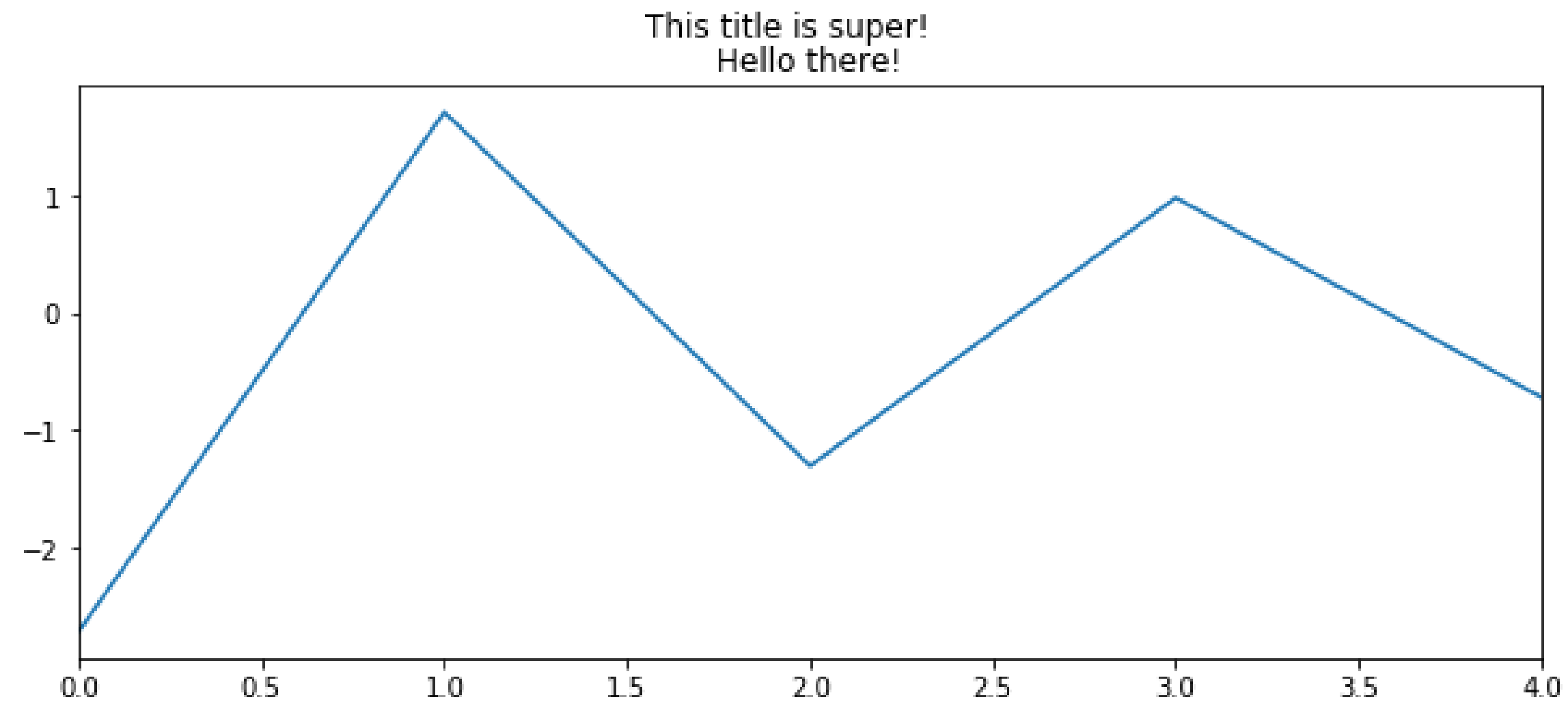
## OPTION 1: PANDAS RETURNS AXIS

```
ax = df_demo["C"].plot(figsize=(10, 4))  
ax.set_title("Hello there!");  
fig = ax.get_figure()  
fig.suptitle("This title is super!");
```



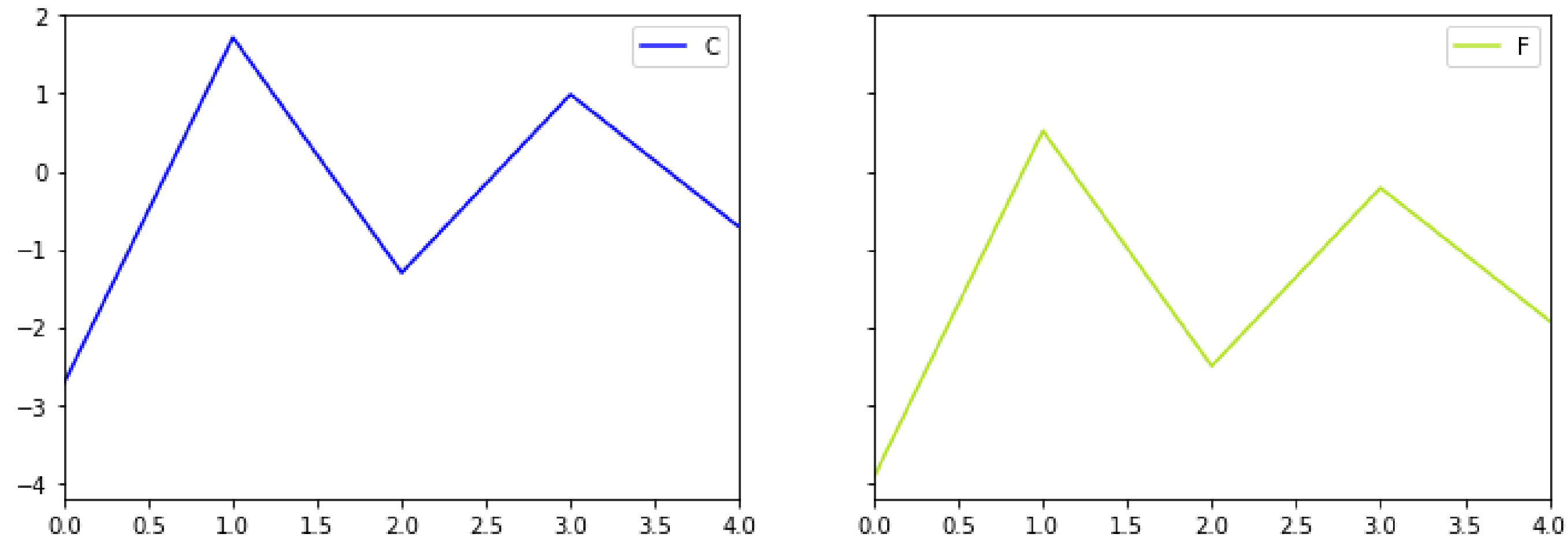
## OPTION 2: DRAW ON MATPLOTLIB AXES

```
fig, ax = plt.subplots(figsize=(10, 4))  
df_demo["C"].plot(ax=ax)  
ax.set_title("Hello there!");  
fig.suptitle("This title is super!");
```



- We can also get fancy!

```
fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=True, figsize=(12, 4))
for ax, column, color in zip([ax1, ax2], ["C", "F"], ["blue", "#b2e123"]):
    df_demo[column].plot(ax=ax, legend=True, color=color)
```

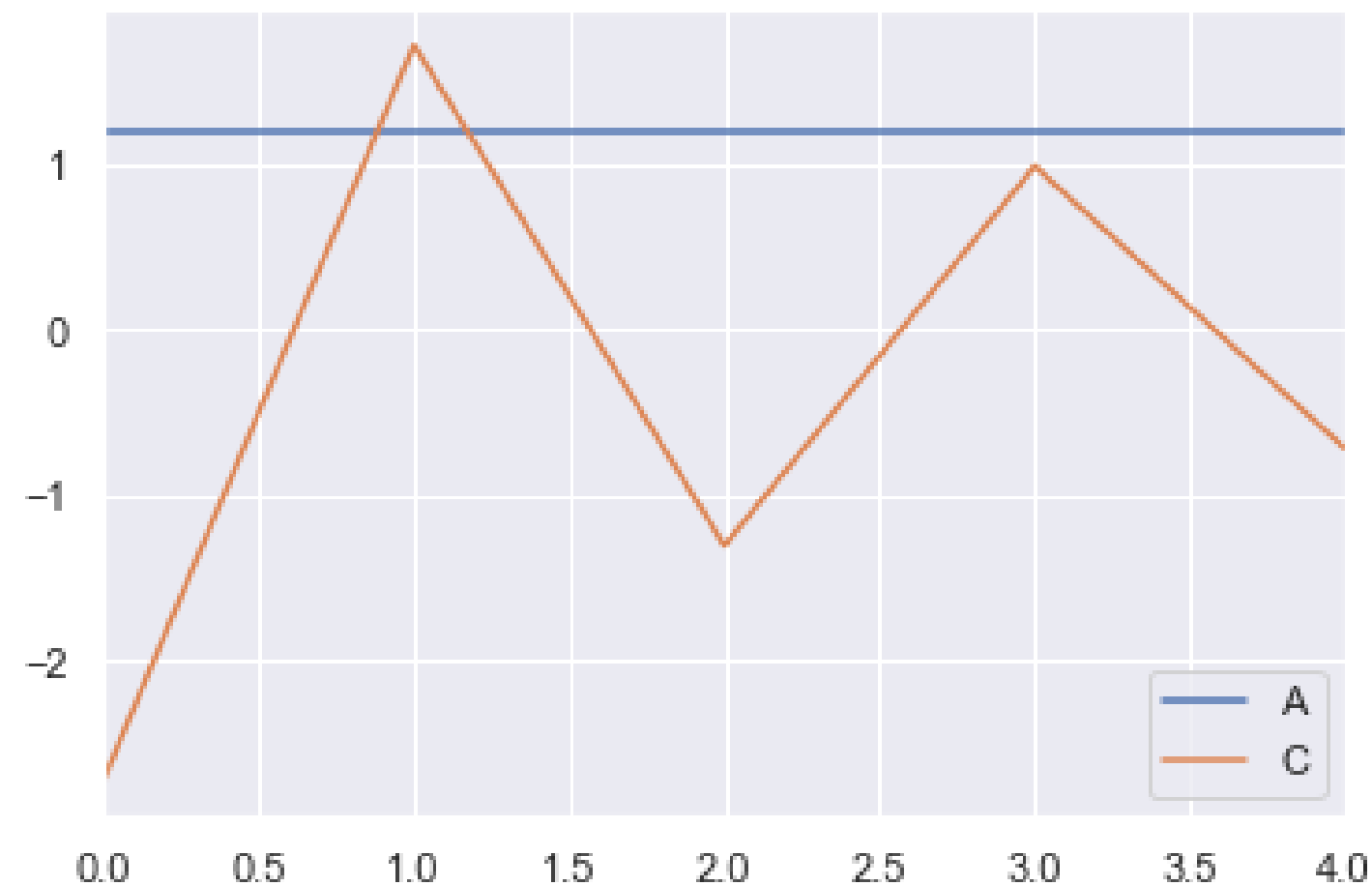


## ASIDE: SEABORN

- Python package on top of Matplotlib
- Powerful API shortcuts for plotting of statistical data
- Manipulate color palettes
- Works well together with Pandas
- Also: New, well-looking defaults for Matplotlib (IMHO)
- → <https://seaborn.pydata.org/>

```
import seaborn as sns
sns.set()
```

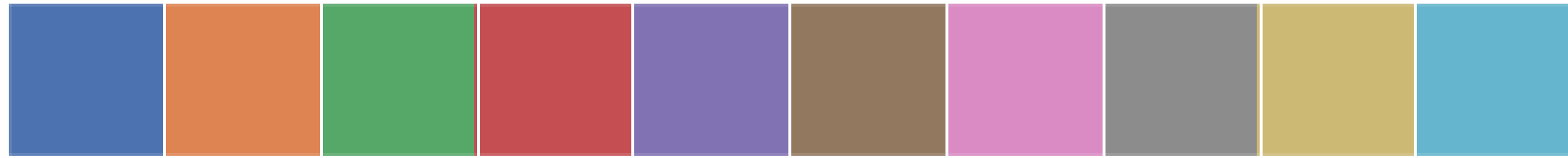
```
df_demo[["A", "C"]].plot();
```



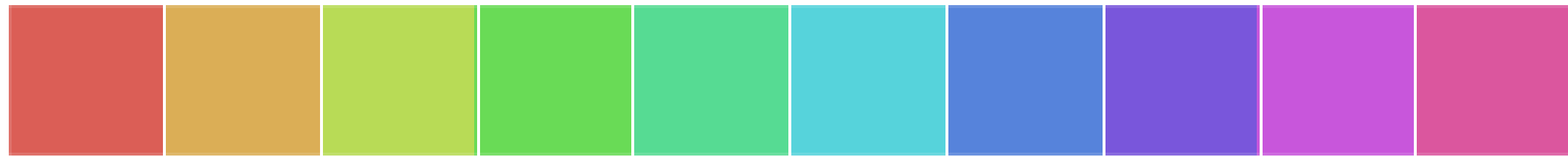
# SEABORN COLOR PALETTE EXAMPLE

- [Documentation](#)

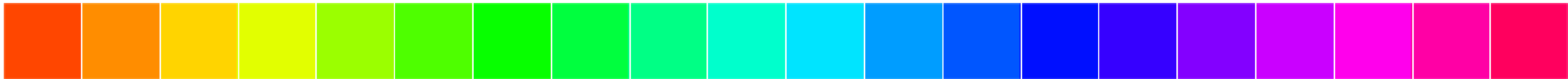
```
sns.palplot(sns.color_palette())
```



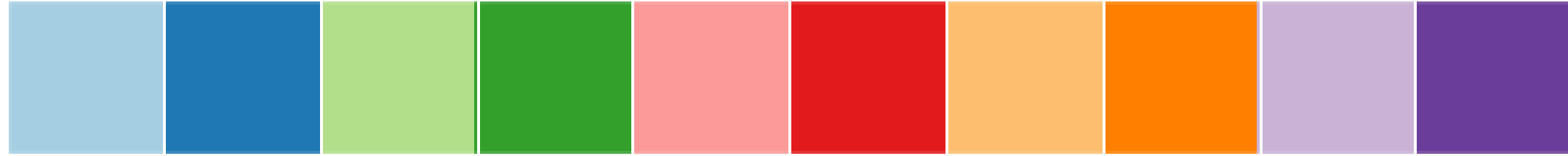
```
sns.palplot(sns.color_palette("hls", 10))
```



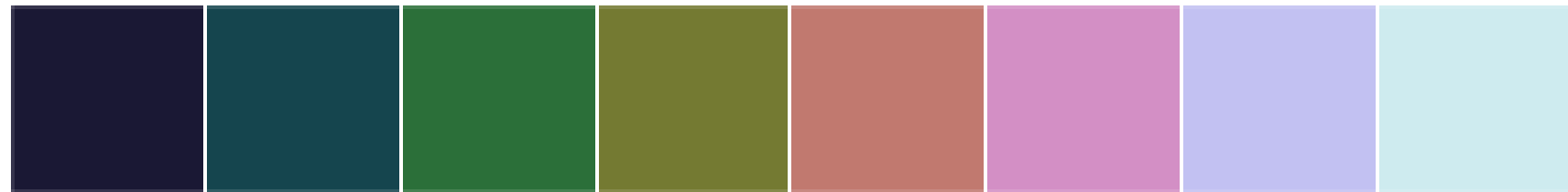
```
sns.palplot(sns.color_palette("hsv", 20))
```



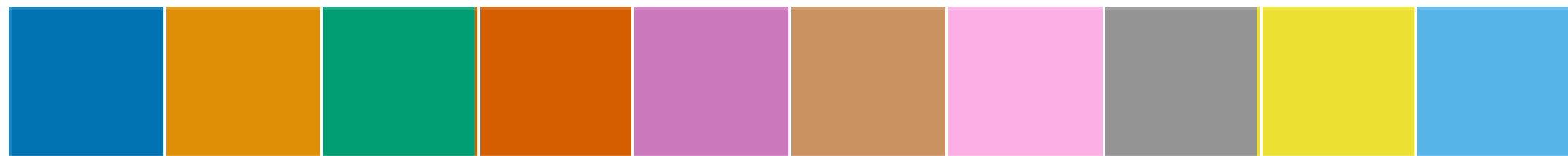
```
sns.palplot(sns.color_palette("Paired", 10))
```



```
sns.palplot(sns.color_palette("cubehelix", 8))
```



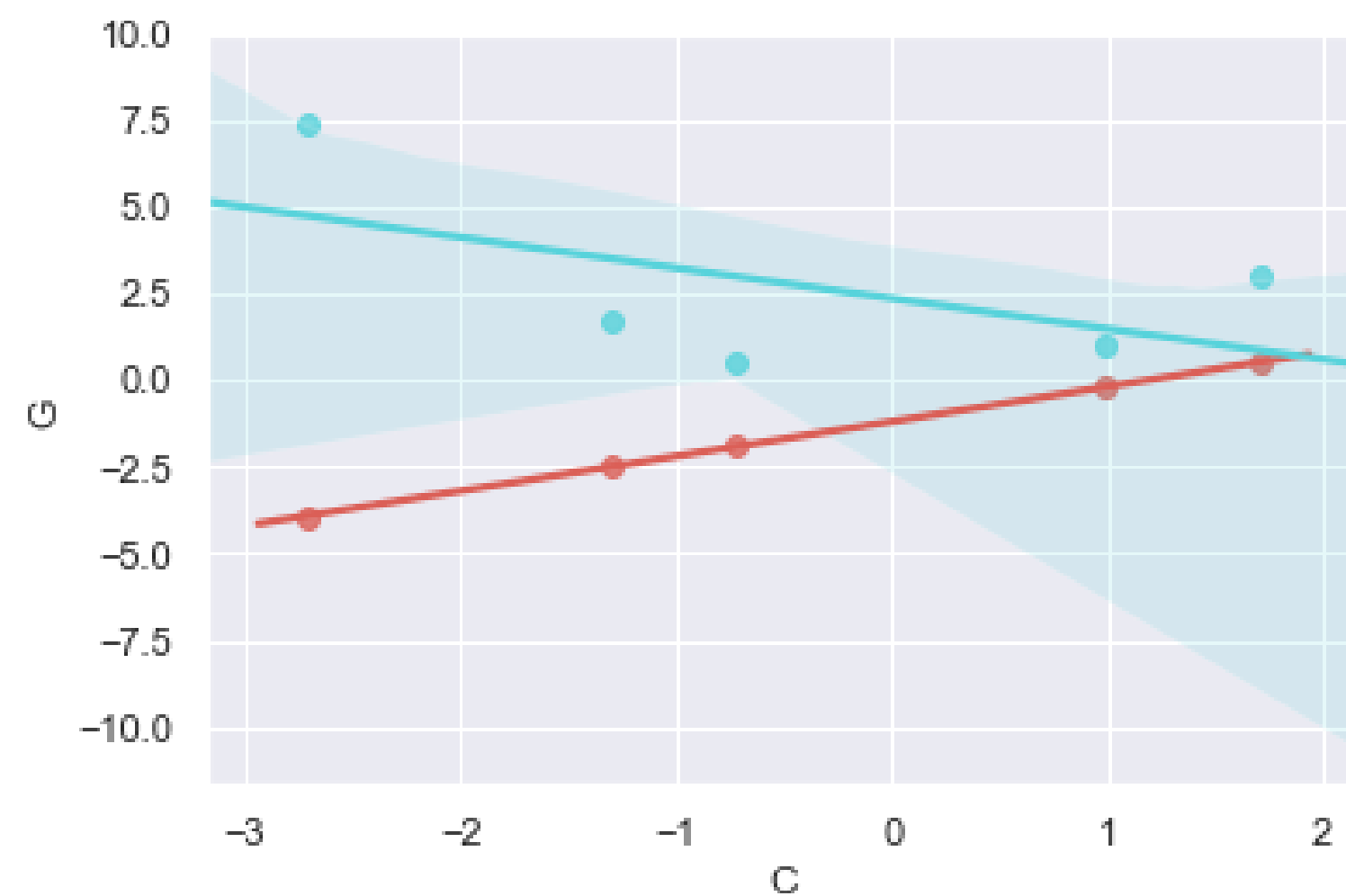
```
sns.palplot(sns.color_palette("colorblind", 10))
```



# SEABORN PLOT EXAMPLES

- Most of the time, I use a regression plot from Seaborn

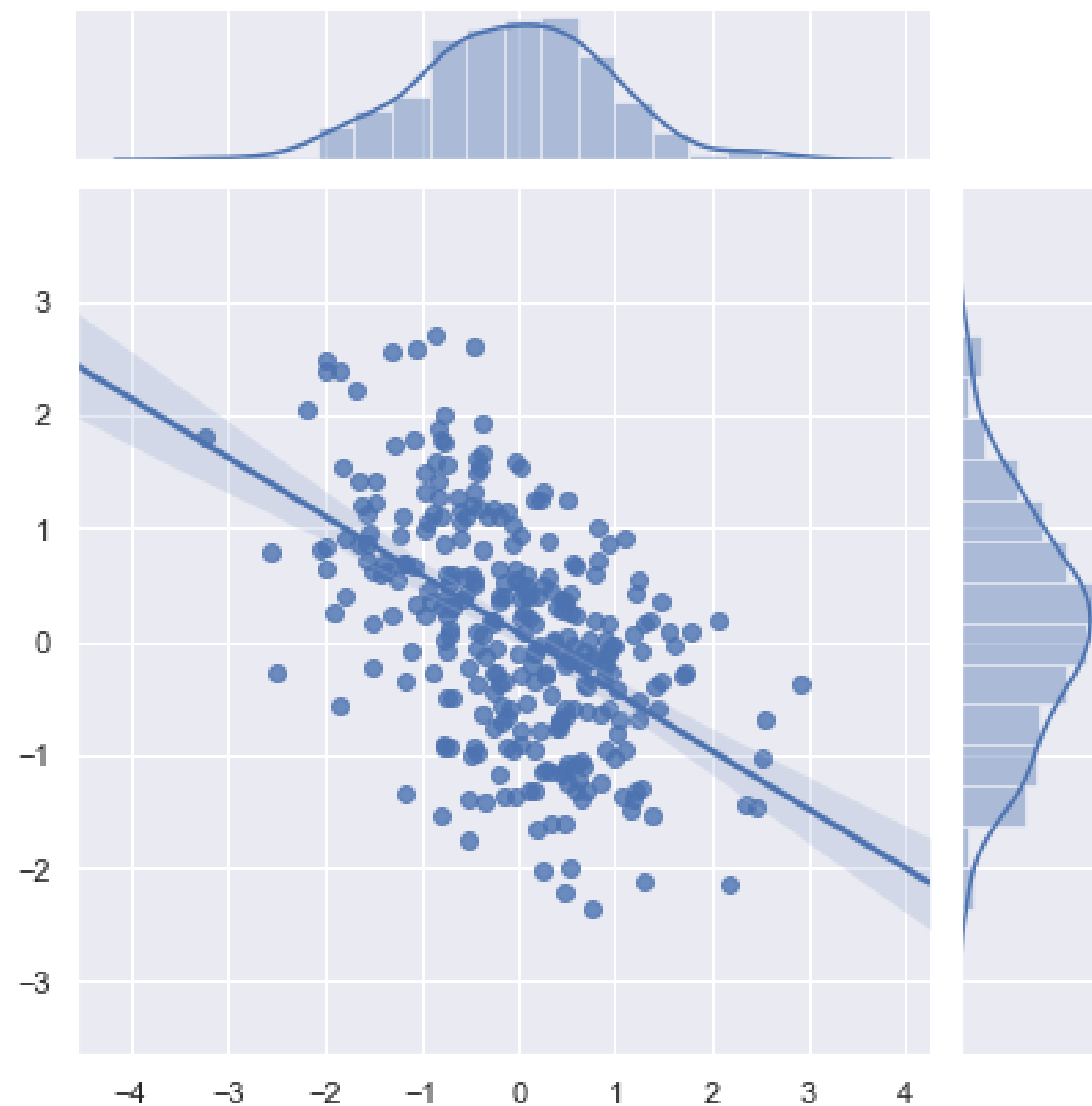
```
with sns.color_palette("hls", 2):  
    sns.regplot(x="C", y="F", data=df_demo);  
    sns.regplot(x="C", y="G", data=df_demo);
```



- A *joint plot* combines two plots relating to distribution of values into one
- Very handy for showing a fuller picture of two-dimensionally scattered variables

```
x, y = np.random.multivariate_normal([0, 0], [[1, -.5], [-.5, 1]], size=300).T
```

```
sns.jointplot(x=x, y=y, kind="reg");
```



## TASK 6

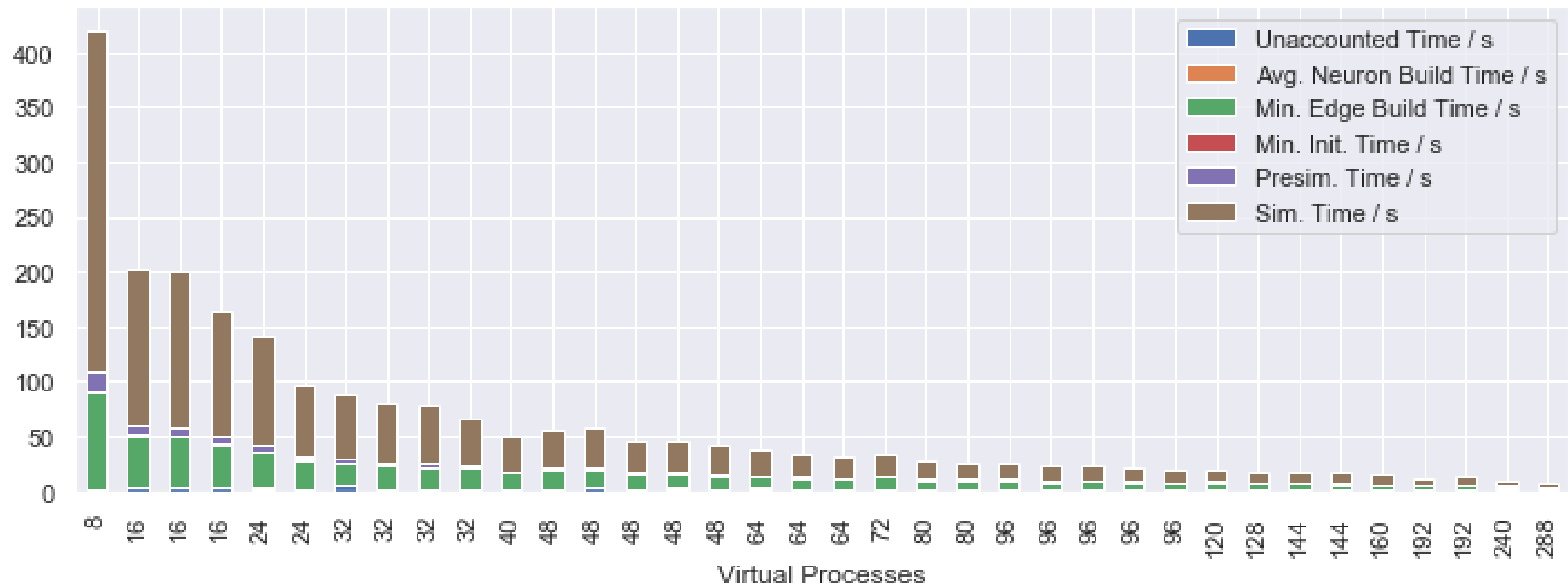
- To your `df` NEST data frame, add a column with the unaccounted time (`Unaccounted Time / s`), which is the difference of program runtime, average neuron build time, minimal edge build time, minimal initialization time, presimulation time, and simulation time.  
(I know this is technically not super correct, but it will do for our example.)
- Plot a stacked bar plot of all these columns (except for program runtime) over the virtual processes
- Remember: [pollev.com/aherten538](https://pollev.com/aherten538)

```
cols = [  
    'Avg. Neuron Build Time / s',  
    'Min. Edge Build Time / s',  
    'Min. Init. Time / s',  
    'Presim. Time / s',  
    'Sim. Time / s'  
]  
df["Unaccounted Time / s"] = df['Runtime Program / s']  
for entry in cols:  
    df["Unaccounted Time / s"] = df["Unaccounted Time / s"] - df[entry]
```

```
df[["Runtime Program / s", "Unaccounted Time / s", *cols]].head(2)
```

	Runtime Program / s	Unaccounted Time / s	Avg. Neuron Build Time / s	Min. Edge Build Time / s	Min. Init. Time / s	Presim. Time / s	Sim. Time / s
Virtual Processes							
8	420.42	2.09	0.29	88.12	1.14	17.26	311.52
16	202.15	2.43	0.28	47.98	0.70	7.95	142.81

```
df[["Unaccounted Time / s", *cols]].plot(kind="bar", stacked=True, figsize=(12, 4));
```

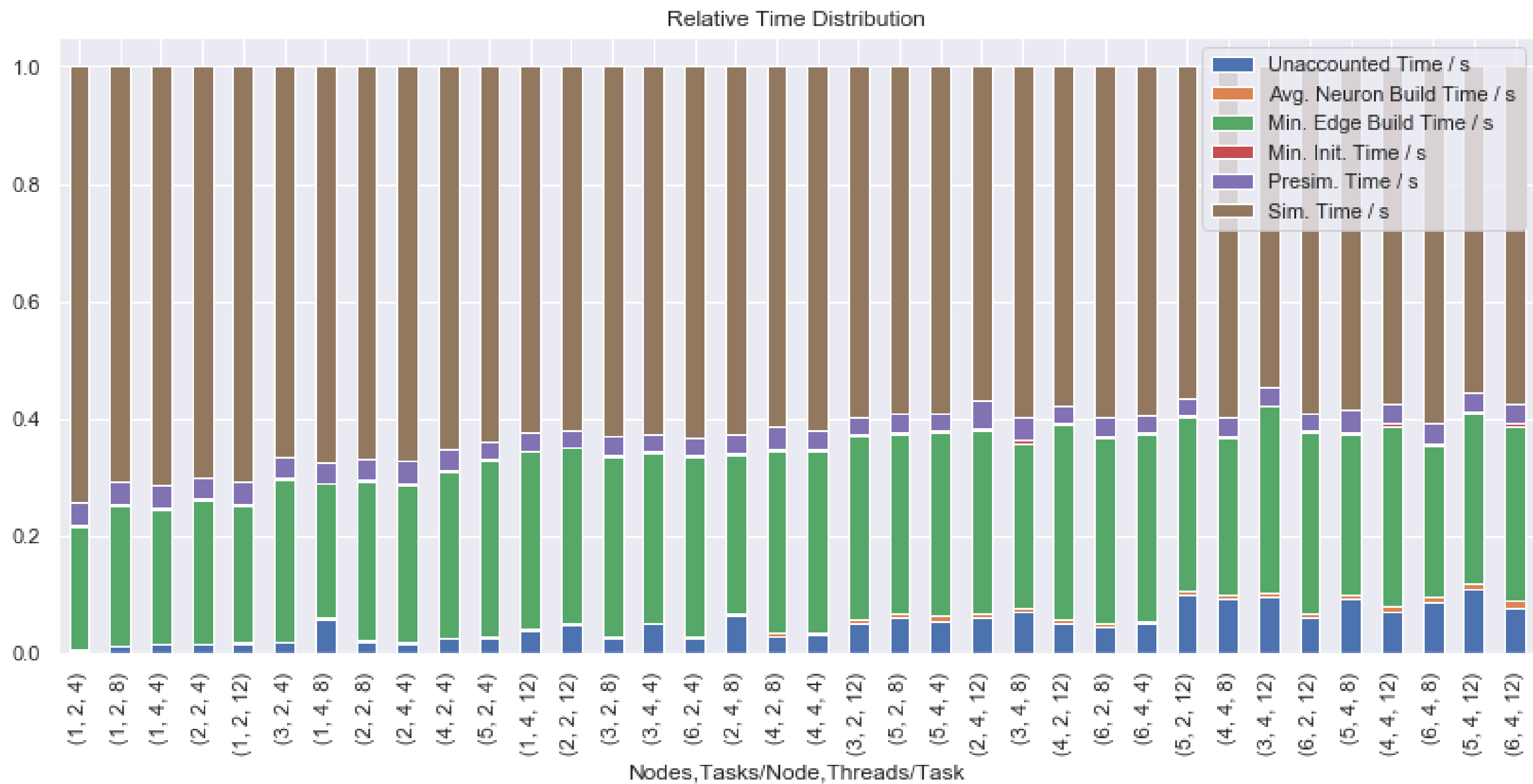


- Make it relative to the total program run time
- **Slight complication:** Our virtual processes as indexes are not unique; we need to find new unique indexes
- Let's use a multi index!

```
df_multind = df.set_index(["Nodes", "Tasks/Node", "Threads/Task"])
df_multind.head()
```

			id	Runtime Program /s	Scale	Plastic	Avg. Neuron Build Time / s	Min. Edge Build Time / s	Max. Edge Build Time / s	Min. Init. Time / s	Max. Init. Time / s	Presim. Time / s	Sim. Time / s	Virt. Memory (Sum) / kB	Local Spike Counter (Sum)	Average Rate (Sum)	Num Neur
Nodes	Tasks/Node	Threads/Task															
1	2	4	5	420.42	10	True	0.29	88.12	88.18	1.14	1.20	17.26	311.52	46560664.0	825499	7.48	1125
		8	5	202.15	10	True	0.28	47.98	48.48	0.70	1.20	7.95	142.81	47699384.0	802865	7.03	1125
	4	4	5	200.84	10	True	0.15	46.03	46.34	0.70	1.01	7.87	142.97	46903088.0	802865	7.03	1125
2	2	4	5	164.16	10	True	0.20	40.03	41.09	0.52	1.58	6.08	114.88	46937216.0	802865	7.03	1125
1	2	12	6	141.70	10	True	0.30	32.93	33.26	0.62	0.95	5.41	100.16	50148824.0	813743	7.27	1125

```
df_multind[["Unaccounted Time / s", *cols]]\  
    .divide(df_multind["Runtime Program / s"], axis="index")\  
    .plot(kind="bar", stacked=True, figsize=(14, 6), title="Relative Time Distribution");
```



# NEXT LEVEL: HIERARCHICAL DATA

- MultiIndex only a first level
- More powerful:
  - Grouping: `.groupby()` ([API](#))
  - Pivoting: `.pivot_table()` ([API](#)); also `.pivot()` ([API](#))

```
df.groupby("Nodes").mean()
```

	id	Tasks/Node	Threads/Task	Runtime Program / s	Scale	Plastic	Avg. Neuron Build Time / s	Min. Edge Build Time / s	Max. Edge Build Time / s	Min. Init. Time / s	...	Presim. Time / s	Sim. Time / s	Virt. Me (Sun
Nodes														
1	5.333333	3.0	8.0	185.023333	10.0	True	0.220000	42.040000	42.838333	0.583333	...	7.226667	132.061667	4.806581
2	5.333333	3.0	8.0	73.601667	10.0	True	0.168333	19.628333	20.313333	0.191667	...	2.725000	48.901667	4.975288
3	5.333333	3.0	8.0	43.990000	10.0	True	0.138333	12.810000	13.305000	0.135000	...	1.426667	27.735000	5.511161
4	5.333333	3.0	8.0	31.225000	10.0	True	0.116667	9.325000	9.740000	0.088333	...	1.066667	19.353333	5.325781
5	5.333333	3.0	8.0	24.896667	10.0	True	0.140000	7.468333	7.790000	0.070000	...	0.771667	14.950000	6.075634
6	5.333333	3.0	8.0	20.215000	10.0	True	0.106667	6.165000	6.406667	0.051667	...	0.630000	12.271667	6.060651

6 rows × 21 columns

# PIVOTING

- Combine categorically-similar columns
- Creates hierarchical index
- Respected during plotting!
- A pivot table has three *layers*; if confused, think about these questions
  - `index`: »What's on the `x` axis?«
  - `values`: »What value do I want to plot?«
  - `columns`: »What categories do I want [to be in the legend]?«
- All can be populated from base data frame
- Might be aggregated, if needed

```
df_demo["H"] = [(-1)**n for n in range(5)]
```

```
df_pivot = df_demo.pivot_table(
    index="F",
    values="G",
    columns="H"
)
df_pivot
```

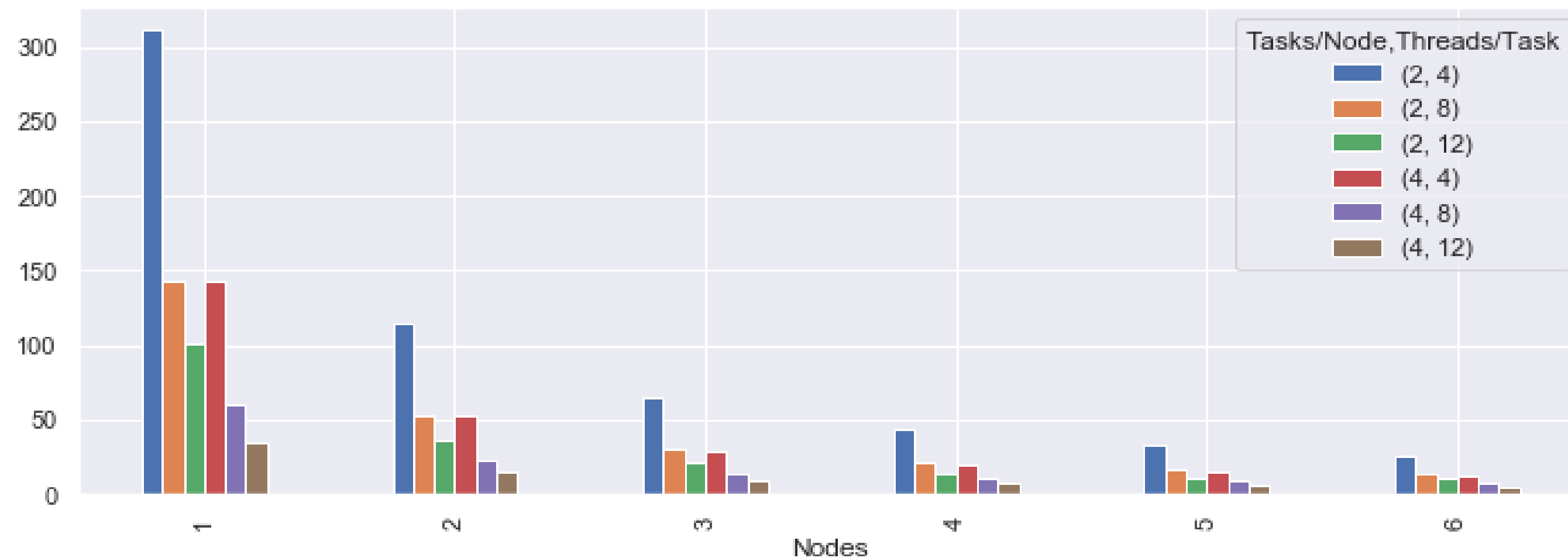
	H	-1	1
F			
-3.918282	NaN	7.389056	
-2.504068	NaN	1.700594	
-1.918282	NaN	0.515929	
-0.213769	0.972652	NaN	
0.518282	2.952492	NaN	

```
df_pivot.plot();
```

# TASK 7

- Create a pivot table based on the NEST `df` data frame
- Let the `x` axis show the number of nodes; display the values of the simulation time `"Sim. Time / s"` for the tasks per node and threas per task configurations
- Please plot a bar plot
- Done? [pollev.com/aherten538](https://pollev.com/aherten538)

```
df.pivot_table(  
    index=["Nodes"],  
    columns=["Tasks/Node", "Threads/Task"],  
    values="Sim. Time / s",  
).plot(kind="bar", figsize=(12, 4));
```



# THE END

- Pandas works on data frames
- Slice frames to your likings
- Plot frames
  - Together with Matplotlib, Seaborn, others
- Pivot tables are next level greatness
- Remember: ***Pandas as early as possible!***
- Thanks for being here! 🥰

Tell me what you think about this tutorial! [a.herten@fz-juelich.de](mailto:a.herten@fz-juelich.de)

Next slide: Further reading

# FURTHER READING

- [Pandas User Guide](#)
- [Matplotlib and LaTeX Plots](#)
- towardsdatascience.com:
  - [Pandas DataFrame: A lightweight Intro](#)
  - [Introduction to Data Visualization in Python](#)
  - [Basic Time Series Manipulation with Pandas](#)
  - [An Introduction to Scikit Learn: The Gold Standard of Python Machine Learning](#)
  - [Mapping with Matplotlib, Pandas, Geopandas and Basemap in Python](#)

# POLL RESULTS

