

DATA ANALYSIS AND PLOTTING IN PYTHON WITH PANDAS

Andreas Herten, Jülich Supercomputing Centre, Forschungszentrum Jülich, 12 May 2022

MY MOTIVATION

- I like Python
- I like plotting data
- I like sharing
- I think Pandas is awesome and you should use it too
- ...*but I'm no Python expert!*

Motto: »Pandas as early as possible!«

TASK OUTLINE

- Task 1
- Task 2
- Task 3
- Task 4
- Task 5
- Task 6
- Task 7
- Task 7B

TUTORIAL SETUP

- 3½ hours, including break around 10:30
- Alternating between lecture and hands-on
- Please give status of hands-ons via 👍 as BigBlueButton status
- Please now open Jupyter Notebook of this session: <https://go.fzj.de/jsc-pd22>
- Give thumbs up! 👍

ABOUT PANDAS

- Python package (~~Python 2~~, Python 3)
- For data analysis and manipulation
- 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`
- Cheatsheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf



PANDAS COHABITATION

- Pandas works great together with other established Python tools
 - [Jupyter Notebooks](#)
 - Plotting with `matplotlib`
 - Numerical analysis with `numpy`
 - Modelling with `statsmodels`, `scikit-learn`
 - Nicer plots with `seaborn`, `altair`, `plotly`
 - Performance enhancement with [Cython](#), [Numba](#), ...
- Tools building up on Pandas: [cuDF](#) (GPU-accelerated DataFrames in [Rapids](#)), ...

FIRST STEPS

```
In [7]: import pandas
```

```
In [8]: import pandas as pd
```

```
In [9]: pd.__version__
```

```
Out[9]: '1.4.2'
```

```
In [10]: %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
```

```
-----
```

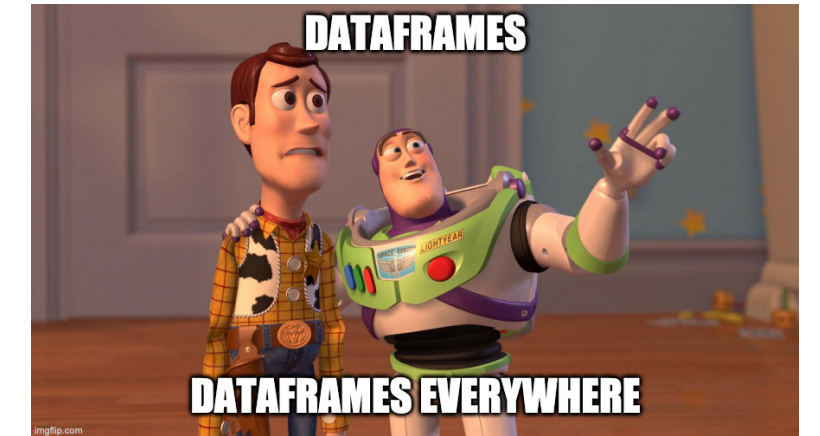
```
Here are just a few of the things that pandas does well:
```

- Easy handling of missing data in floating point as well as non-floating point data.
- Size mutability: columns can be inserted and deleted from DataFrame & higher dimensional objects
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let `'Series'`, `'DataFrame'`, etc. automatically align the data for you in computations.
- Powerful, flexible group by functionality to perform split-apply-comb

DATAFRAMES

It's all about DataFrames

- Data containers of Pandas:
 - Linear: `Series`
 - Multi Dimension: `DataFrame`
- `Series` is *only* special (1D) case of `DataFrame`
- → We use `DataFrame`s as the more general case here



DATAFRAMES

Construction

- To show features of `DataFrame`, let's construct one and show by example!
- 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

```
In [11]: ages = [41, 56, 56, 57, 39, 59, 43, 56, 38, 60]
```

```
In [12]: pd.DataFrame(ages)
```

```
Out[12]:
```

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

```
In [13]: df_ages = pd.DataFrame(ages)
df_ages.head(3)
```

```
Out[13]:
```

	0
0	41
1	56
2	56

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

```
In [14]: data = {
          "Name": ["Liu", "Rowland", "Rivers", "Waters", "Rice", "Fields", "Kerr", "Romero", "Davis", "Hall"],
          "Age": ages
        }
print(data)
```

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

```
In [15]: df_sample = pd.DataFrame(data)
df_sample.head(4)
```

```
Out[15]:
```

	Name	Age
0	Liu	41
1	Rowland	56
2	Rivers	56
3	Waters	57

- Automatically creates columns from dictionary
- Two columns now; one for names, one for ages

```
In [16]: df_sample.columns
```

```
Out[16]: Index(['Name', 'Age'], dtype='object')
```

- First column is *index*
- `DataFrame` always have indexes; auto-generated or custom

```
In [17]: df_sample.index
```

Out[17]: `RangeIndex(start=0, stop=10, step=1)`

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

```
In [18]: df_sample.set_index("Name", inplace=True)
df_sample
```

Out[18]:

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

- Some more operations

```
In [19]: df_sample.describe()
```

Out[19]:

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

```
In [20]: df_sample.T
```

Out[20]:

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

```
In [21]: df_sample.T.columns
```

Out[21]: Index(['Liu', 'Rowland', 'Rivers', 'Waters', 'Rice', 'Fields', 'Kerr', 'Romero', 'Davis', 'Hall'], dtype='object', name='Name')

- Also: Arithmetic operations

```
In [22]: df_sample.multiply(2).head(3)
```

Out [22]:

Age	
Name	
Liu	82
Rowland	112
Rivers	112

```
In [23]: df_sample.reset_index().multiply(2).head(3)
```

Out [23]:

	Name	Age
0	LiuLiu	82
1	RowlandRowland	112
2	RiversRivers	112

```
In [24]: (df_sample / 2).head(3)
```

Out [24]:

Age	
Name	
Liu	20.5
Rowland	28.0
Rivers	28.0

```
In [25]: (df_sample * df_sample).head(3)
```

Out[25]:

Age	
Name	
Liu	1681
Rowland	3136
Rivers	3136

```
In [26]: def mysquare(number: float) -> float:
          return number*number

df_sample.apply(mysquare).head()
# or: df_sample.apply(lambda x: x*x).head()
```

Out[26]:

Age	
Name	
Liu	1681
Rowland	3136
Rivers	3136
Waters	3249
Rice	1521

```
In [28]: df_sample.apply(np.square).head()
```

Out[28]:

Age	
Name	
Liu	1681
Rowland	3136
Rivers	3136
Waters	3249
Rice	1521

Logical operations allowed as well

```
In [29]: df_sample > 40
```

Out [29]:

Age	
Name	
Liu	True
Rowland	True
Rivers	True
Waters	True
Rice	False
Fields	True
Kerr	True
Romero	True
Davis	False
Hall	True

```
In [30]: df_sample.apply(mysquare).head() == df_sample.apply(lambda x: x*x).head()
```

Out [30]:

Age	
Name	
Liu	True
Rowland	True
Rivers	True
Waters	True
Rice	True

- Create data frame with
 - 6 names of dinosaurs,
 - their favourite prime number,
 - and their favorite color.
- Play around with the frame
- Tell me when you're done with status icon in BigBlueButton: 👍

```
In [31]: happy_dinos = {
    "Dinosaur Name": [],
    "Favourite Prime": [],
    "Favourite Color": []
}
#df_dinos =
```

```
In [32]: 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
```

Out [32]:

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

MORE DataFrame EXAMPLES

```
In [33]: 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
```

Out [33]:

	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

```
In [34]: df_demo.sort_values("C")
```

Out [34]:

	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

In [35]: `df_demo.round(2).tail(2)`

Out[35]:

	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

In [36]: `df_demo.round(2).sum()`

`/var/folders/f5/swf8tg5j5r7bqbwftt3zn6b00000gn/T/ipykernel_67517/1325867503.py:1: FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError. Select only valid columns before calling the reduction.`
`df_demo.round(2).sum()`

Out[36]:

A	6.0
C	-2.03
E	SameSameSameSameSame

dtype: object

In [37]: `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}
```

`/var/folders/f5/swf8tg5j5r7bqbwftt3zn6b00000gn/T/ipykernel_67517/3396683912.py:1: FutureWarning: The method `DataFrame.to_latex` is deprecated and will be removed in a future version. The arguments signature may therefore change. It is recommended to use `DataFrame.to_latex` which also contains additional functionality.`
`print(df_demo.round(2).to_latex())`

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]
}
```

```
In [38]: pd.read_json("data-lost.json").set_index("Character").sort_index()
```

Out [38]:

	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

- Read in `data-nest.csv` to `DataFrame` ; call it `df`
(Data was produced with *JUBE*)
- Get to know it and play a bit with it
- Tell me when you're done with status icon in BigBlueButton: 👍

```
In [39]: !head data-nest.csv
```

```
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
5,1,2,8,202.15,10,true,0.28,47.98,48.48,0.70,1.20,7.95,142.81,47699384.00,802865,7.03,112500,1265738500,1.5,1.5
5,1,4,8,89.57,10,true,0.15,20.41,23.21,0.23,3.04,3.19,60.31,46813040.00,821491,7.23,112500,1265738500,1.5,1.5
5,2,2,4,164.16,10,true,0.20,40.03,41.09,0.52,1.58,6.08,114.88,46937216.00,802865,7.03,112500,1265738500,1.5,1.5
5,2,4,4,77.68,10,true,0.13,20.93,21.22,0.16,0.46,3.12,52.05,47362064.00,821491,7.23,112500,1265738500,1.5,1.5
5,2,2,8,79.60,10,true,0.20,21.63,21.91,0.19,0.47,2.98,53.12,46847168.00,821491,7.23,112500,1265738500,1.5,1.5
5,2,4,8,37.20,10,true,0.13,10.08,11.60,0.10,1.63,1.24,23.29,47065232.00,818198,7.33,112500,1265738500,1.5,1.5
5,3,2,4,96.51,10,true,0.15,26.54,27.41,0.36,1.22,3.33,64.28,52256880.00,813743,7.27,112500,1265738500,1.5,1.5
```

```
In [40]: df = pd.read_csv("data-nest.csv")
df.head()
```

Out[40]:

	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	Local
0	5	1	2	4	420.42	10	True	0.29	88.12	88.18	...	1.20	
1	5	4	4	4	200.84	10	True	0.15	46.03	46.34	...	1.01	
2	5	1	2	8	202.15	10	True	0.28	47.98	48.48	...	1.20	

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=<object object>, 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, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None, storage_options=None)
```

SLICING OF DATA FRAMES

- Slicing: Select a sub-range / sub-set of entire data frame
- Pandas documentation: [Detailed documentation](#), [short documentation](#)

QUICK SLICES

- Use square-bracket operators to slice data frame quickly: `[]`
 - Use column name to select column
 - Use numerical value to select row
- Example: Select only columnn `C` from `df_demo`

```
In [41]: df_demo.head(3)
```

```
Out[41]:
```

	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

```
In [42]: df_demo['C']
```

```
Out[42]:
```

0	-2.718282
1	1.718282
2	-1.304068
3	0.986231
4	-0.718282

Name: C, dtype: float64

- Instead of column name in quotes and square brackets: Name of column *directly*

```
In [43]: df_demo.C
```

```
Out[43]: 0    -2.718282  
1     1.718282  
2    -1.304068  
3     0.986231  
4    -0.718282  
Name: C, dtype: float64
```

- I'm not a friend, because no spaces allowed
(And Pandas as early as possible means labelling columns well and adding spaces)

- Select more than one column by providing `list` to slice operator `[]`
- Example: Select list of columns `A` and `C`, `['A', 'C']` from `df_demo`

```
In [44]: my_slice = ['A', 'C']
df_demo[my_slice]
```

Out [44]:

	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

- Use numerical values in brackets to slice along rows
- Use ranges just like with Python lists

```
In [45]: df_demo[1:3]
```

Out[45]:

	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

```
In [46]: df_demo[1:6:2]
```

Out[46]:

	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: location might change after re-sorting!

```
In [47]: df_demo[1:3]
```

Out[47]:

	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

```
In [48]: df_demo.sort_values("C")[1:3]
```

Out[48]:

	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

SLICING OF DATA FRAMES

Better Slicing

- `.iloc[]` and `.loc[]` : Faster slicing interfaces with more options

In [49]: `df_demo.iloc[1:3]`

Out [49]:

	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

- Also slice along columns (second argument)

In [50]: `df_demo.iloc[1:3, [0, 2]]`

Out [50]:

	A	C
1	1.2	1.718282
2	1.2	-1.304068

- `.iloc[]` : Slice by position (*numerical/integer*)
- `.loc[]` : Slice by label (*named*)
- See difference with a *proper* index (and not the auto-generated default index from before)

```
In [51]: df_demo_indexed = df_demo.set_index("D")
df_demo_indexed
```

Out[51]:

	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

```
In [52]: df_demo_indexed.loc["entries"]
```

Out[52]:

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

```
In [53]: df_demo_indexed.loc[["has", "entries"], ["A", "C"]]
```

Out[53]:

	A	C
D		
has	1.2	-1.304068
entries	1.2	0.986231
entries	1.2	-0.718282

SLICING OF DATA FRAMES

Advanced Slicing: Logical Slicing

- Slice can also be array of booleans

```
In [54]: df_demo[df_demo["C"] > 0]
```

Out[54]:

	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

```
In [55]: df_demo["C"] > 0
```

Out[55]:

```
0    False
1     True
2    False
3     True
4    False
Name: C, dtype: bool
```

```
In [56]: df_demo[(df_demo["C"] < 0) & (df_demo["D"] == "entries")]
```

Out[56]:

	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](#)

In [57]: `df_demo.head(3)`

Out [57]:

	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

In [58]: `df_demo["F"] = df_demo["C"] - df_demo["A"]`
`df_demo.head(3)`

Out [58]:

	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

- `.insert()` allows to specify position of insertion
- `.shape` gives tuple of size of data frame, `vertical`, `horizontal`

```
In [59]: df_demo.insert(df_demo.shape[1] - 1, "E2", df_demo["C"] ** 2)
df_demo.head(3)
```

Out [59]:

	A	B	C	D	E	E2	F
0	1.2	2018-02-26	-2.718282	This	Same	7.389056	-3.918282
1	1.2	2018-02-26	1.718282	column	Same	2.952492	0.518282
2	1.2	2018-02-26	-1.304068	has	Same	1.700594	-2.504068

```
In [60]: df_demo.tail(3)
```

Out[60]:

	A	B	C	D	E	E2	F
2	1.2	2018-02-26	-1.304068	has	Same	1.700594	-2.504068
3	1.2	2018-02-26	0.986231	entries	Same	0.972652	-0.213769
4	1.2	2018-02-26	-0.718282	entries	Same	0.515929	-1.918282

```
In [69]: 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  
    )
```

/var/folders/f5/swf8tg5j5r7bqbwftt3zn6b00000gn/T/ipykernel_67517/363555535.py:1: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
df_demo.append(
)

Out[69]:

	A	B	C	D	E	E2	F
0	1.2	2018-02-26	-2.718282	This	Same	7.389056	-3.918282
1	1.2	2018-02-26	1.718282	column	Same	2.952492	0.518282
2	1.2	2018-02-26	-1.304068	has	Same	1.700594	-2.504068
3	1.2	2018-02-26	0.986231	entries	Same	0.972652	-0.213769
4	1.2	2018-02-26	-0.718282	entries	Same	0.515929	-1.918282
5	1.3	2018-02-27	-0.777000	has it?	Same	NaN	23.000000

`.append()` seems to be deprecated; one needs to use something like the following in the future:

```
pd.concat(  
    [  
        df_demo,  
        pd.DataFrame({"A": 1.3, "B": pd.Timestamp("2018-02-27"), "C": -0.777, "D": "has it?", "E": "Same", "F": 23}, index=[0])  
    ], ignore_index=True  
)
```

COMBINING FRAMES

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

```
In [64]: df_1 = pd.DataFrame({"Key": ["First", "Second"], "Value": [1, 1]})
df_1
```

Out [64]:

	Key	Value
0	First	1
1	Second	1

```
In [65]: df_2 = pd.DataFrame({"Key": ["First", "Second"], "Value": [2, 2]})
df_2
```

Out [65]:

	Key	Value
0	First	2
1	Second	2

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

```
In [66]: pd.concat([df_1, df_2])
```

Out [66]:

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

- Same, but re-index

```
In [67]: pd.concat([df_1, df_2], ignore_index=True)
```

Out [67]:

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

- Concat, but horizontally

```
In [68]: pd.concat([df_1, df_2], axis=1)
```

Out [68]:

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

- Merge on common column

```
In [69]: pd.merge(df_1, df_2, on="Key")
```

Out [69]:

	Key	Value_x	Value_y
0	First	1	2
1	Second	1	2

- Add a column to the Nest data frame form Task 2 called `Threads` which is the total number of threads across all nodes (i.e. the product of threads per task and tasks per node and nodes)
- Tell me when you're done with status icon in BigBlueButton: 👍

```
In [70]: df["Threads"] = df["Nodes"] * df["Tasks/Node"] * df["Threads/Task"]
df.head()
```

Out[70]:

	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	Number of Connections
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	126573850
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	126573850
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	126573850
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	126573850
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	126573850

5 rows x 22 columns

```
In [71]: df.columns
```

```
Out[71]: 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',
        'Sim. Time / s', 'Virt. Memory (Sum) / kB', 'Local Spike Counter (Sum)',
        'Average Rate (Sum)', 'Number of Neurons', 'Number of Connections',
        'Min. Delay', 'Max. Delay', 'Threads'],
        dtype='object')
```

ASIDE: PLOTTING WITHOUT PANDAS

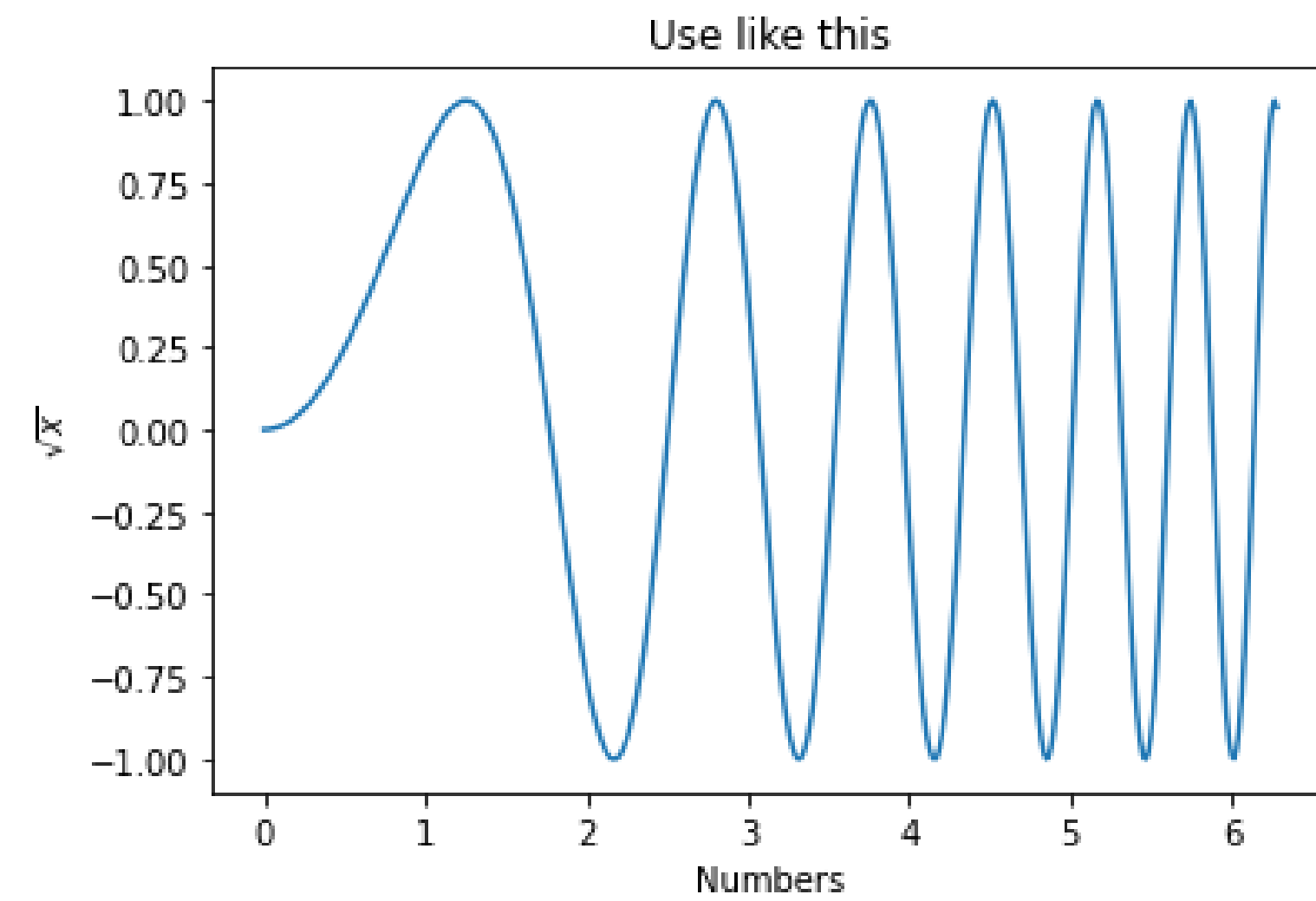
Matplotlib 101

- 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/>

```
In [72]: import matplotlib.pyplot as plt
          %matplotlib inline
```

```
In [73]: x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)
```

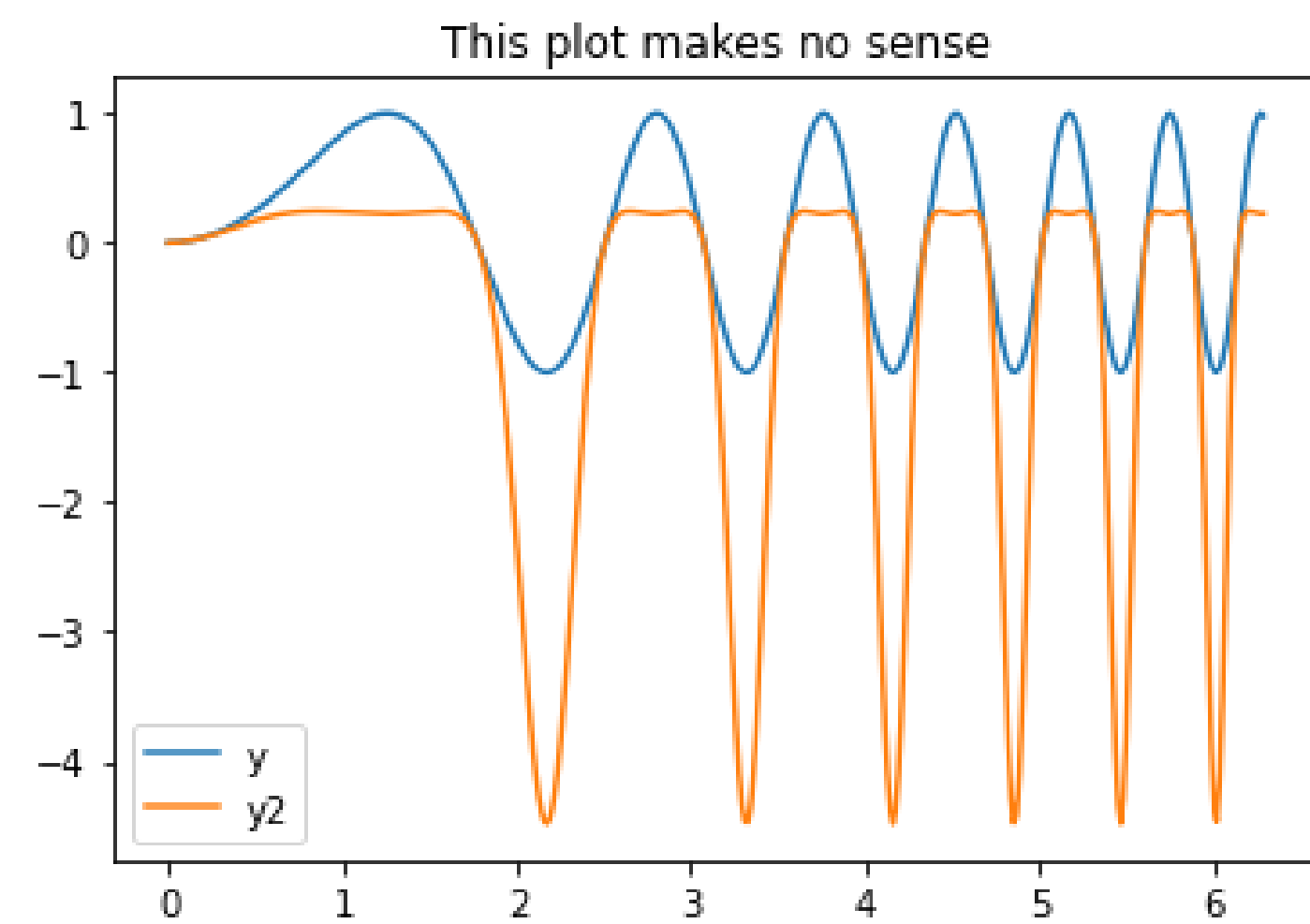
```
In [74]: fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Use like this')
ax.set_xlabel("Numbers");
ax.set_ylabel("$\sqrt{x}$");
```



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

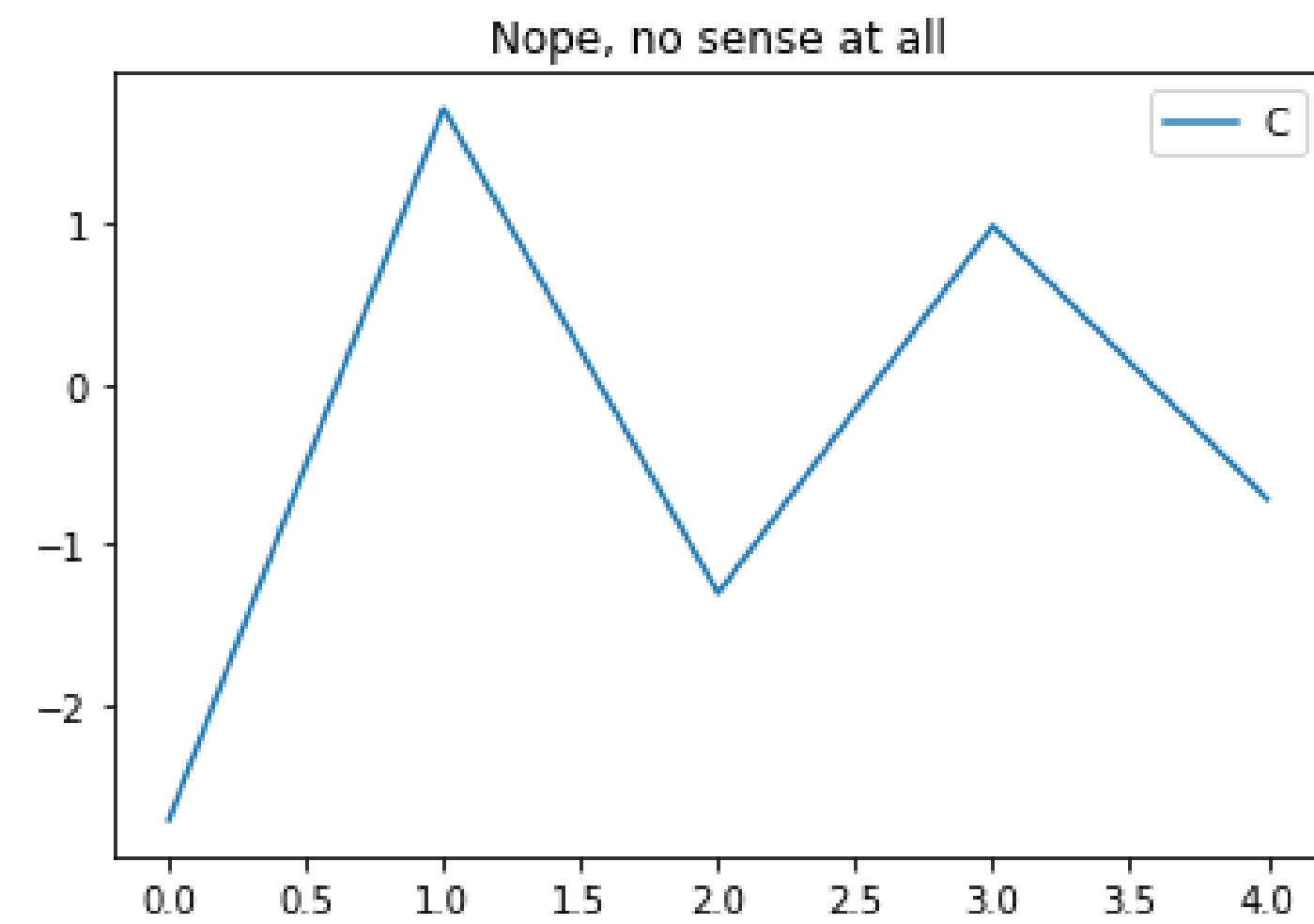
```
In [75]: y2 = y/np.exp(y*1.5)
```

```
In [76]: 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");
```



- Matplotlib can also plot DataFrame data
- Because DataFrame data is *only* array-like data with stuff on top

```
In [77]: fig, ax = plt.subplots()
ax.plot(df_demo.index, df_demo["C"], label="C")
ax.legend()
ax.set_title("Nope, no sense at all");
```



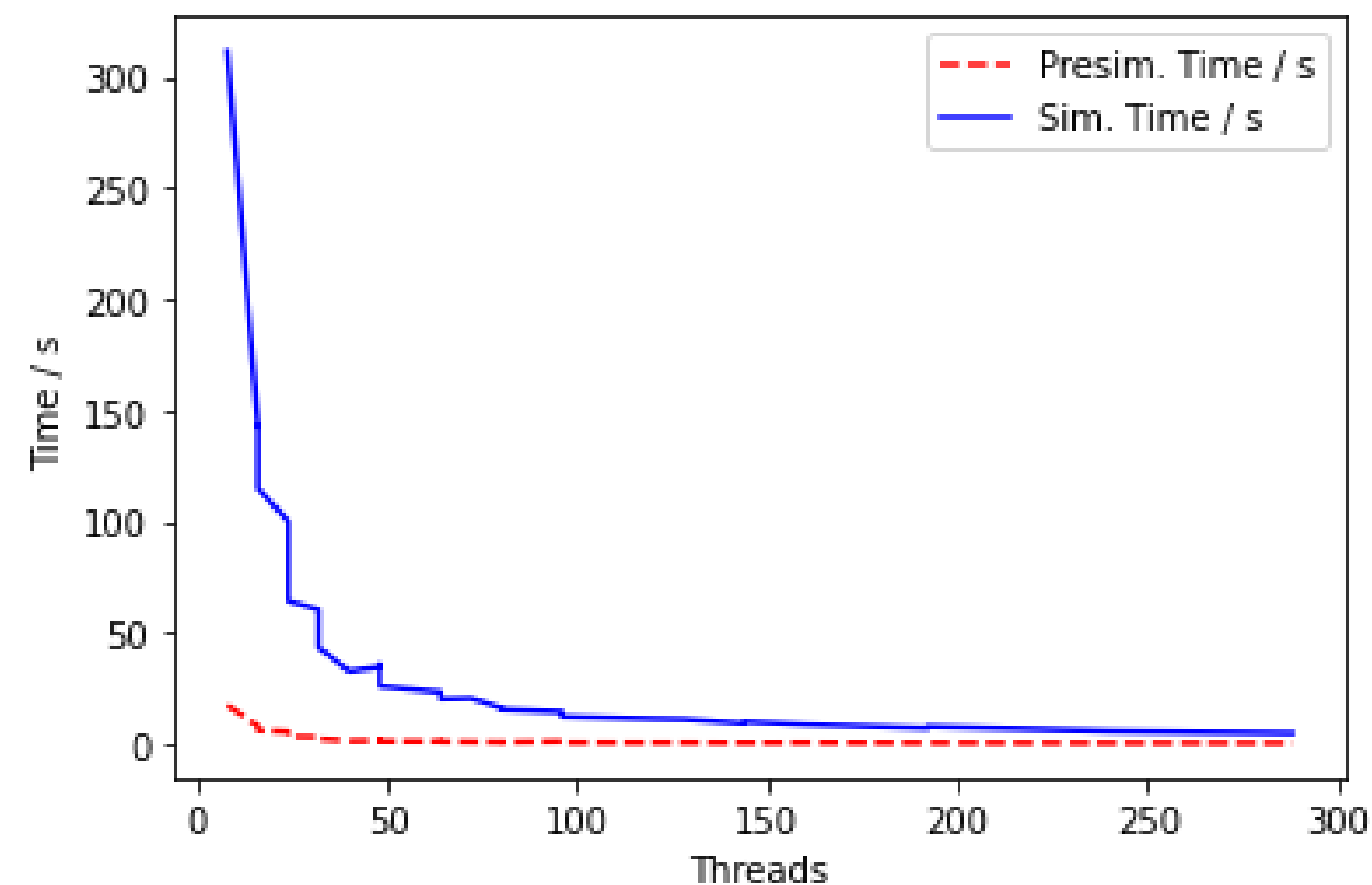
TASK 4

TASK

- Sort the Nest data frame by threads
- Plot "Presim. Time / s" and "Sim. Time / s" of our data frame `df` as a function of threads
- 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 (*1st rule of plotting*)
- Tell me when you're done with status icon in BigBlueButton: 👍

```
In [78]: df.sort_values(["Threads", "Nodes", "Tasks/Node", "Threads/Task"], inplace=True) # multi-level sort
```

```
In [79]: fig, ax = plt.subplots()
ax.plot(df["Threads"], df["Presim. Time / s"], linestyle="dashed", color="red", label="Presim. Time / s")
ax.plot(df["Threads"], df["Sim. Time / s"], "-b", label="Sim. Time / s")
ax.set_xlabel("Threads")
ax.set_ylabel("Time / s")
ax.legend(loc='best');
```

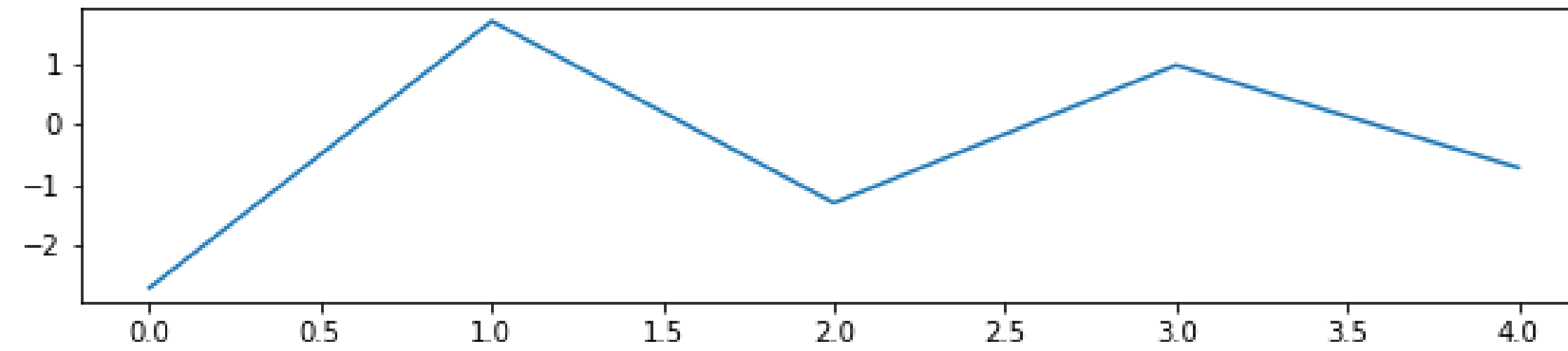


PLOTTING WITH PANDAS

- 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`: Non-parsed keyword passed to Matplotlib's plotting methods

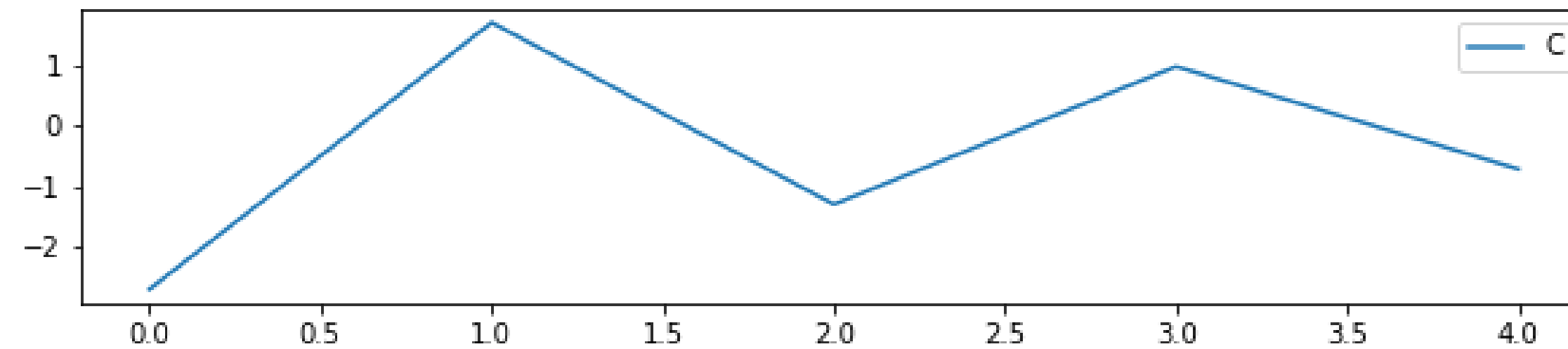
- Either slice and plot...

```
In [80]: df_demo["C"].plot(figsize=(10, 2));
```



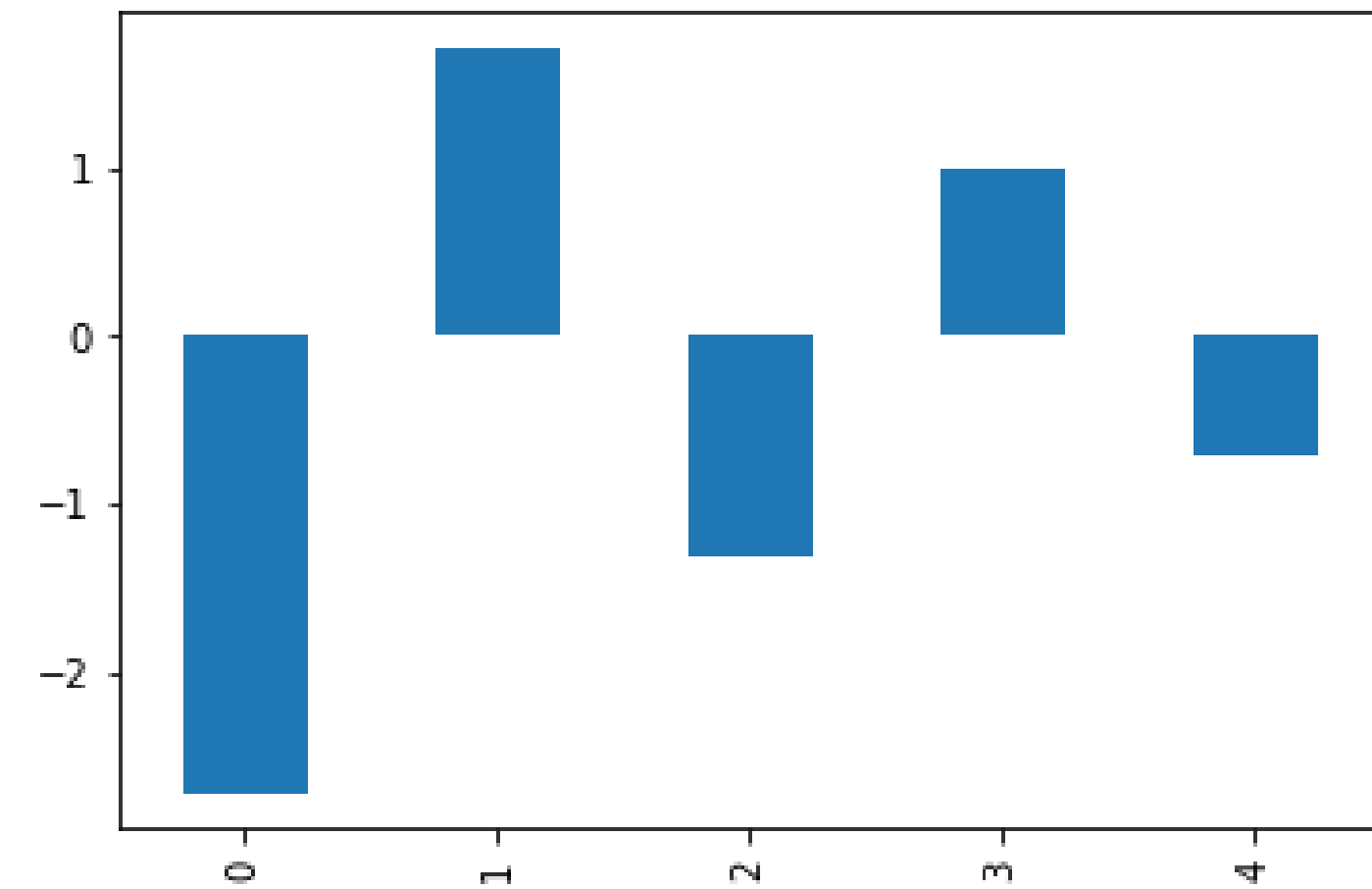
- ... or plot and select

```
In [81]: df_demo.plot(y="C", figsize=(10, 2));
```



- I prefer slicing first:
 - Allows for further operations on the sliced data frame

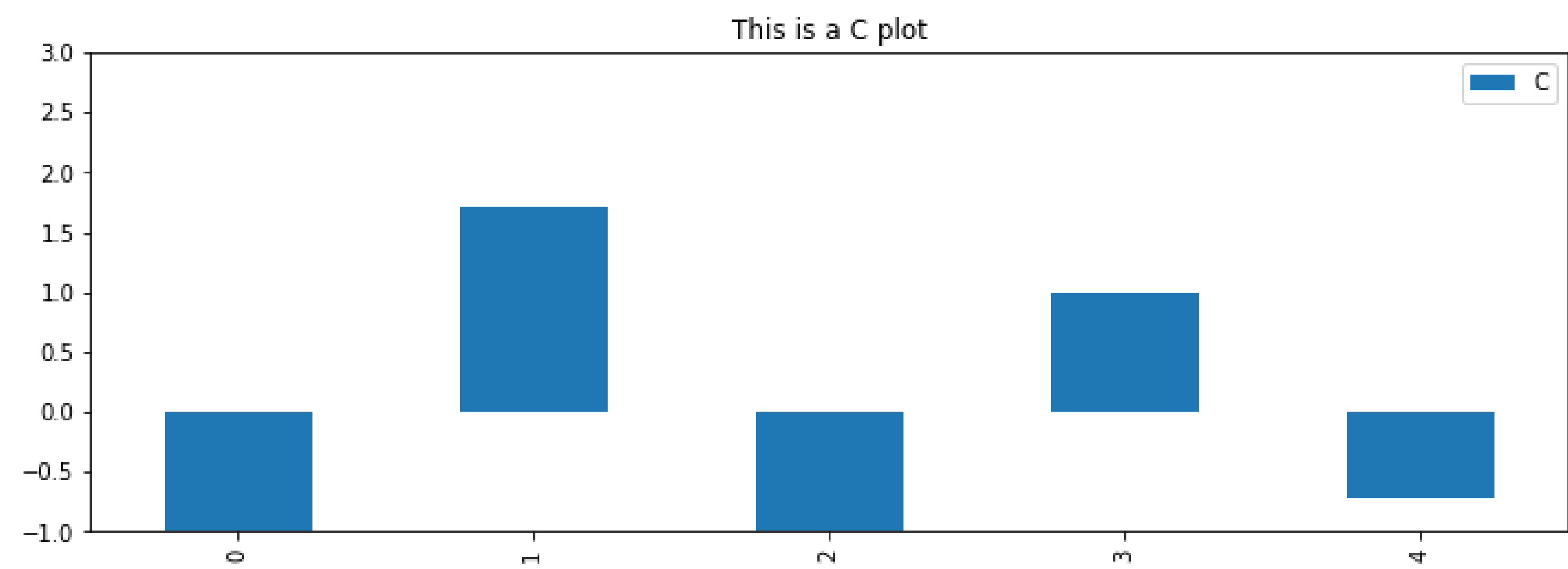
```
In [82]: df_demo["C"].plot(kind="bar");
```



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

```
In [83]: df_demo["C"].plot.bar();
```

```
In [84]: df_demo["C"].plot(kind="bar", legend=True, figsize=(12, 4), ylim=(-1, 3), title="This is a C plot");
```



TASK 5

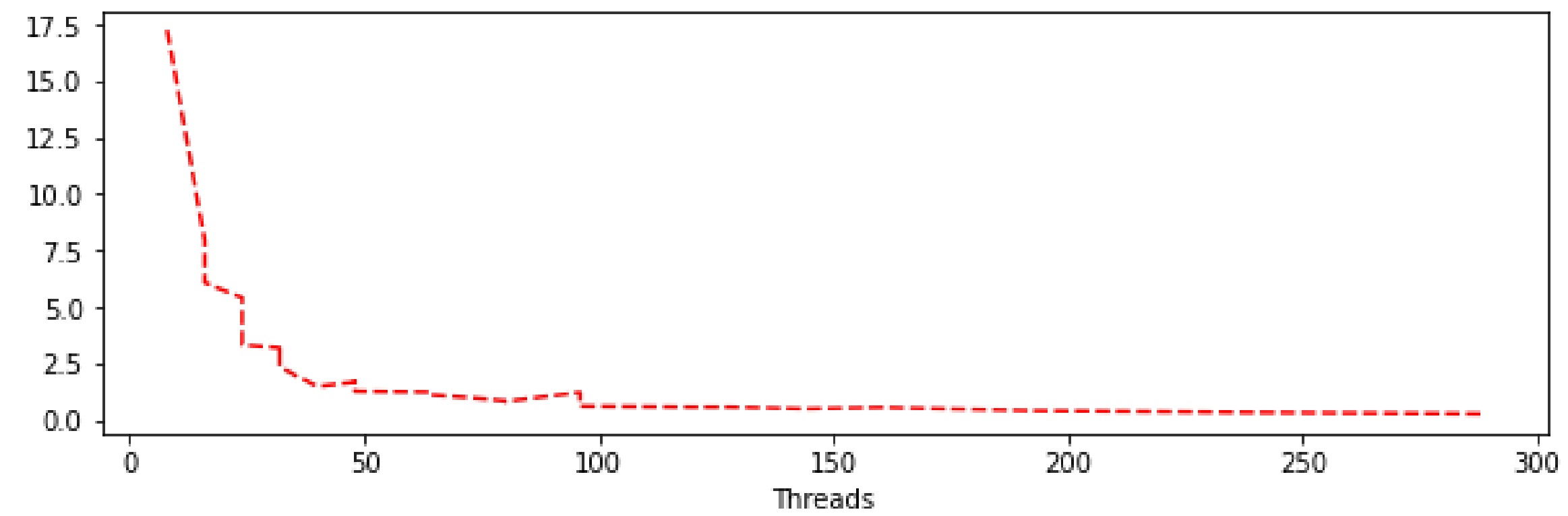
TASK

Use the Nest data frame `df` to:

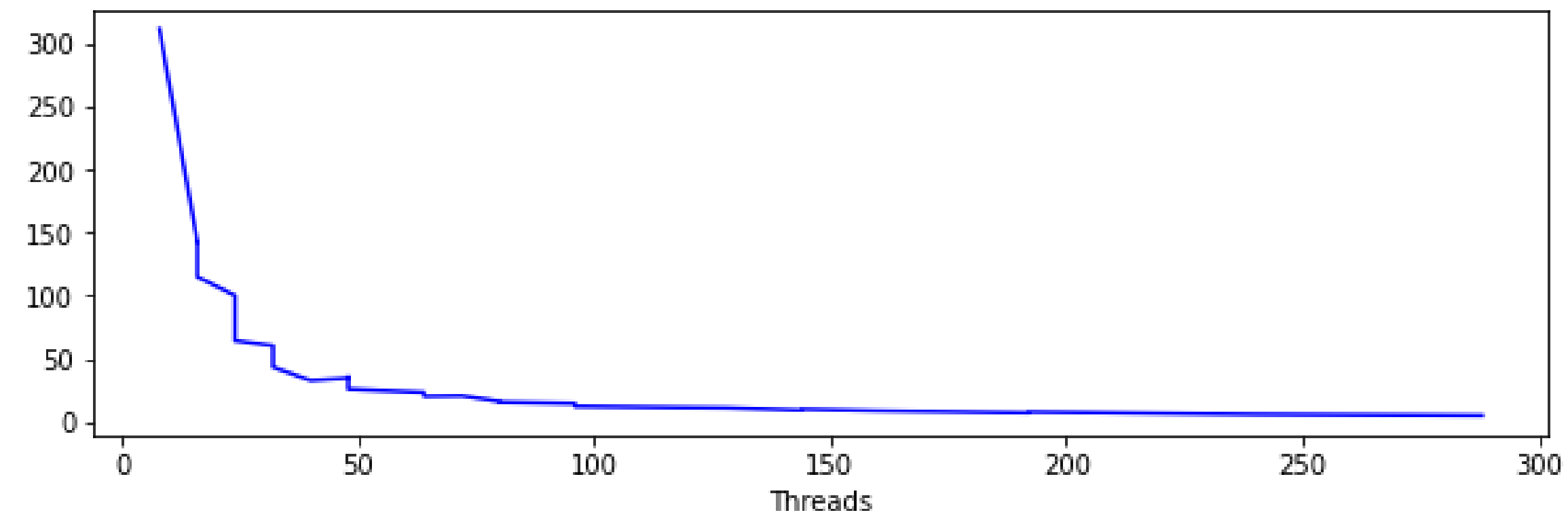
1. Make threads index of the data frame (`.set_index()`)
2. Plot `"Presim. Time / s"` and `"Sim. Time / s"` individually
3. Plot them onto one common canvas!
4. Make them have the same line colors and styles as before
5. Add a legend, add missing axes labels
6. Tell me when you're done with status icon in BigBlueButton: 👍

```
In [85]: df.set_index("Threads", inplace=True)
```

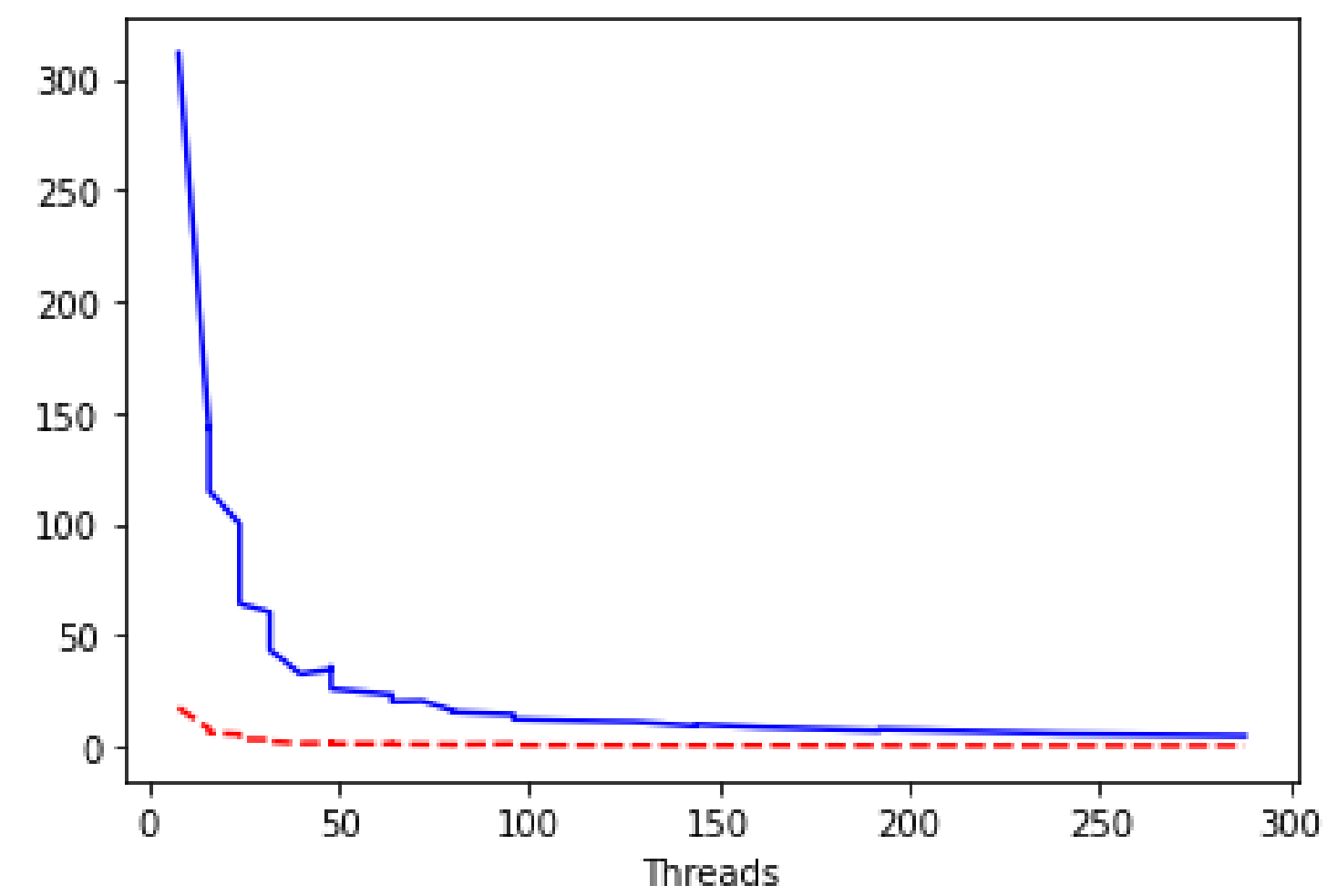
```
In [86]: df["Presim. Time / s"].plot(figsize=(10, 3), style="--", color="red");
```



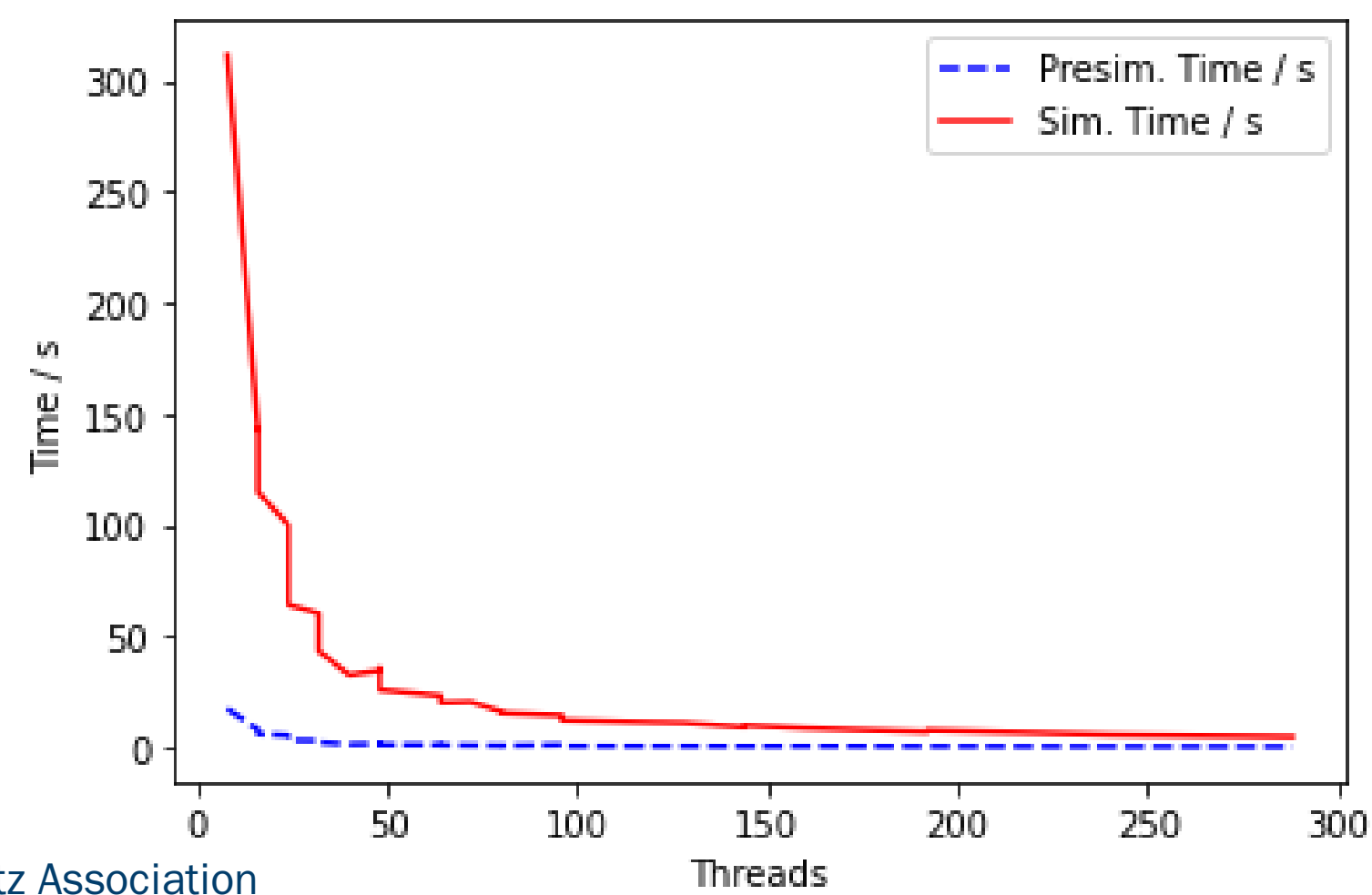
```
In [87]: df["Sim. Time / s"].plot(figsize=(10, 3), style="-b");
```



```
In [88]: df["Presim. Time / s"].plot(style="--r");  
df["Sim. Time / s"].plot(style="-b");
```



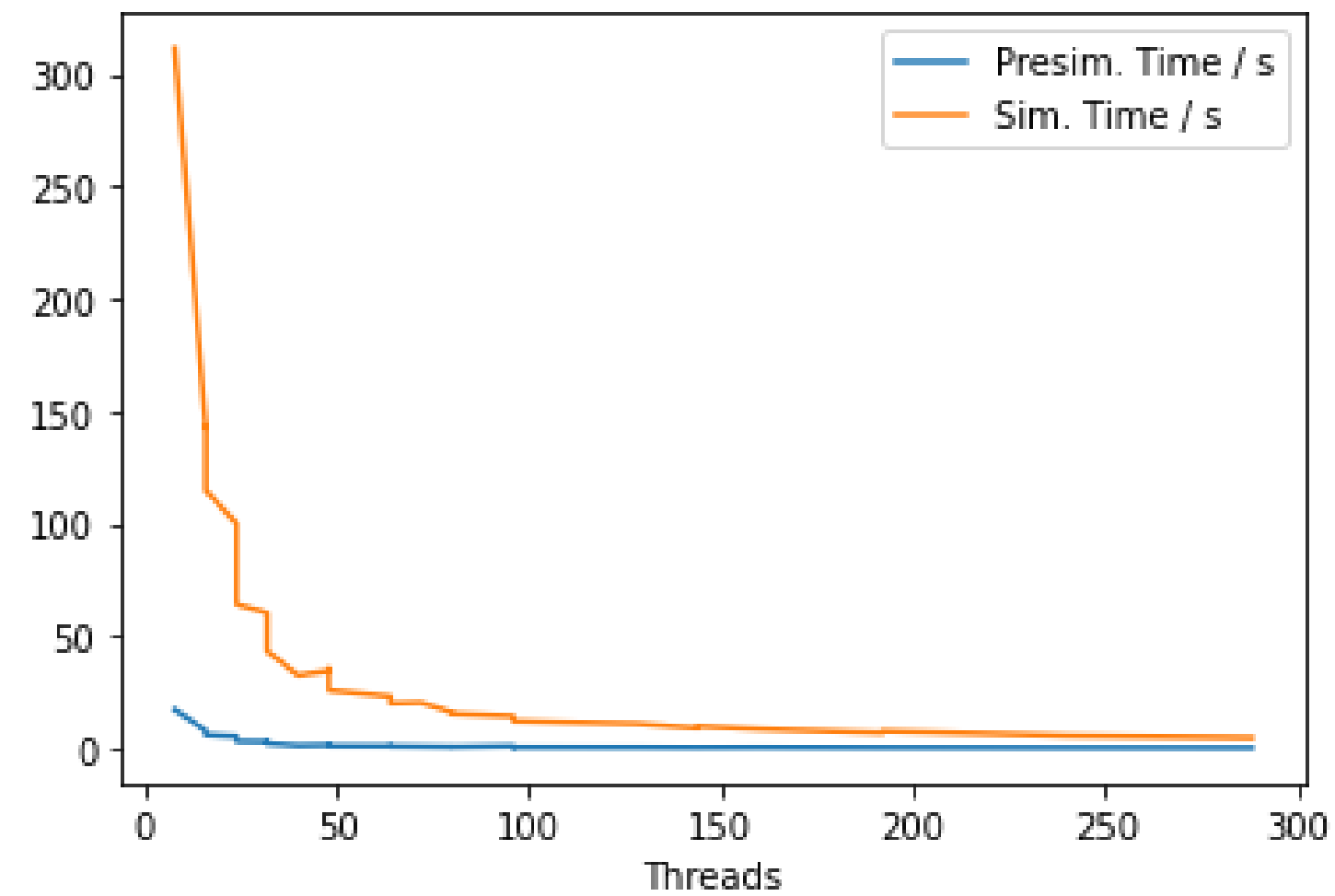
```
In [89]: ax = df[["Presim. Time / s", "Sim. Time / s"]].plot(style=["--b", "-r"]);  
ax.set_ylabel("Time / s");
```



MORE PLOTTING WITH PANDAS

Recap: Our first proper Pandas plot

```
In [90]: df[["Presim. Time / s", "Sim. Time / s"]].plot();
```

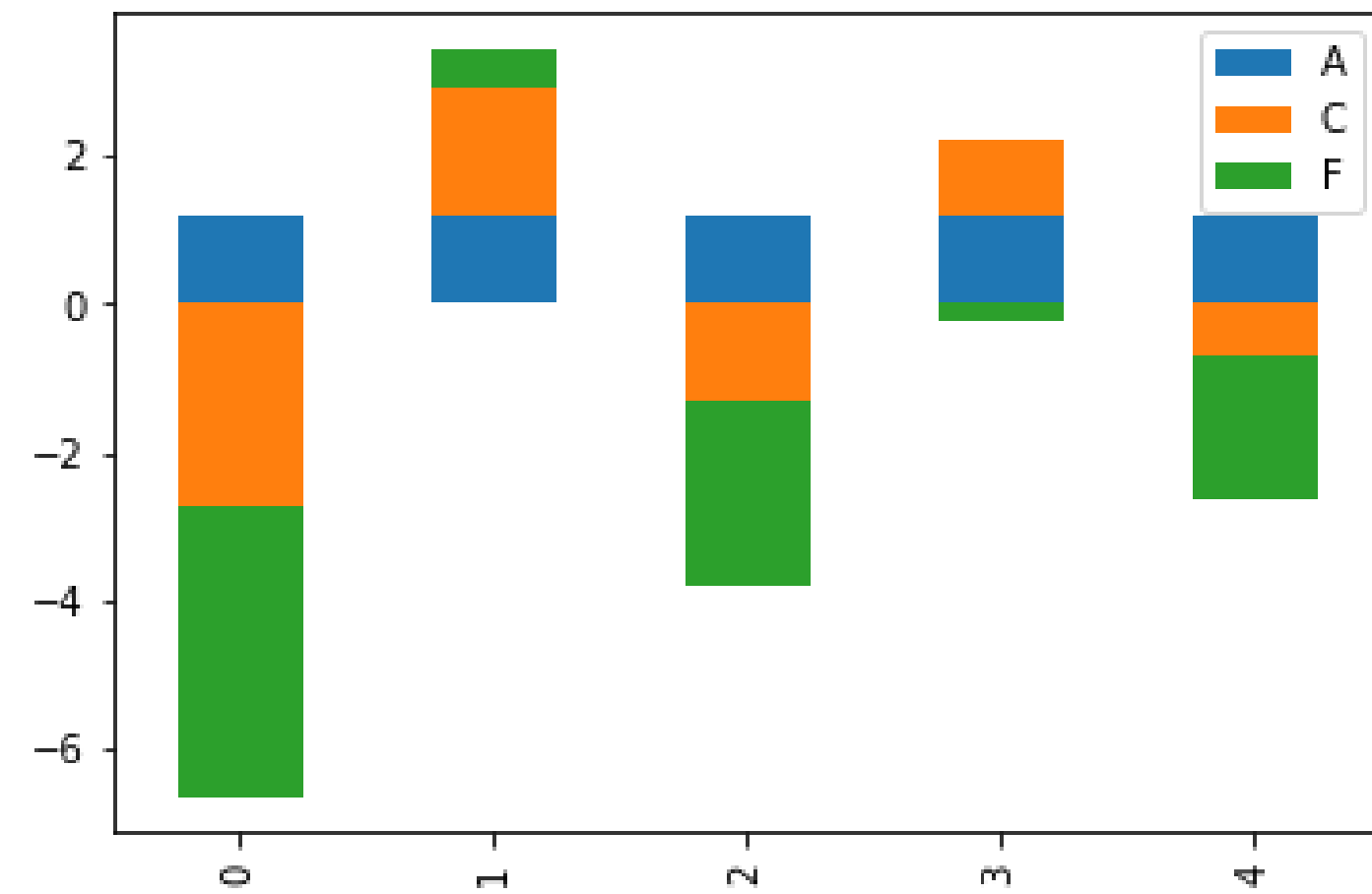


- That's why I think Pandas is great!
- It has great defaults to quickly plot data; basically publication-grade already
- Plotting functionality is very versatile
- Before plotting, data can be *massaged* within data frames, if needed

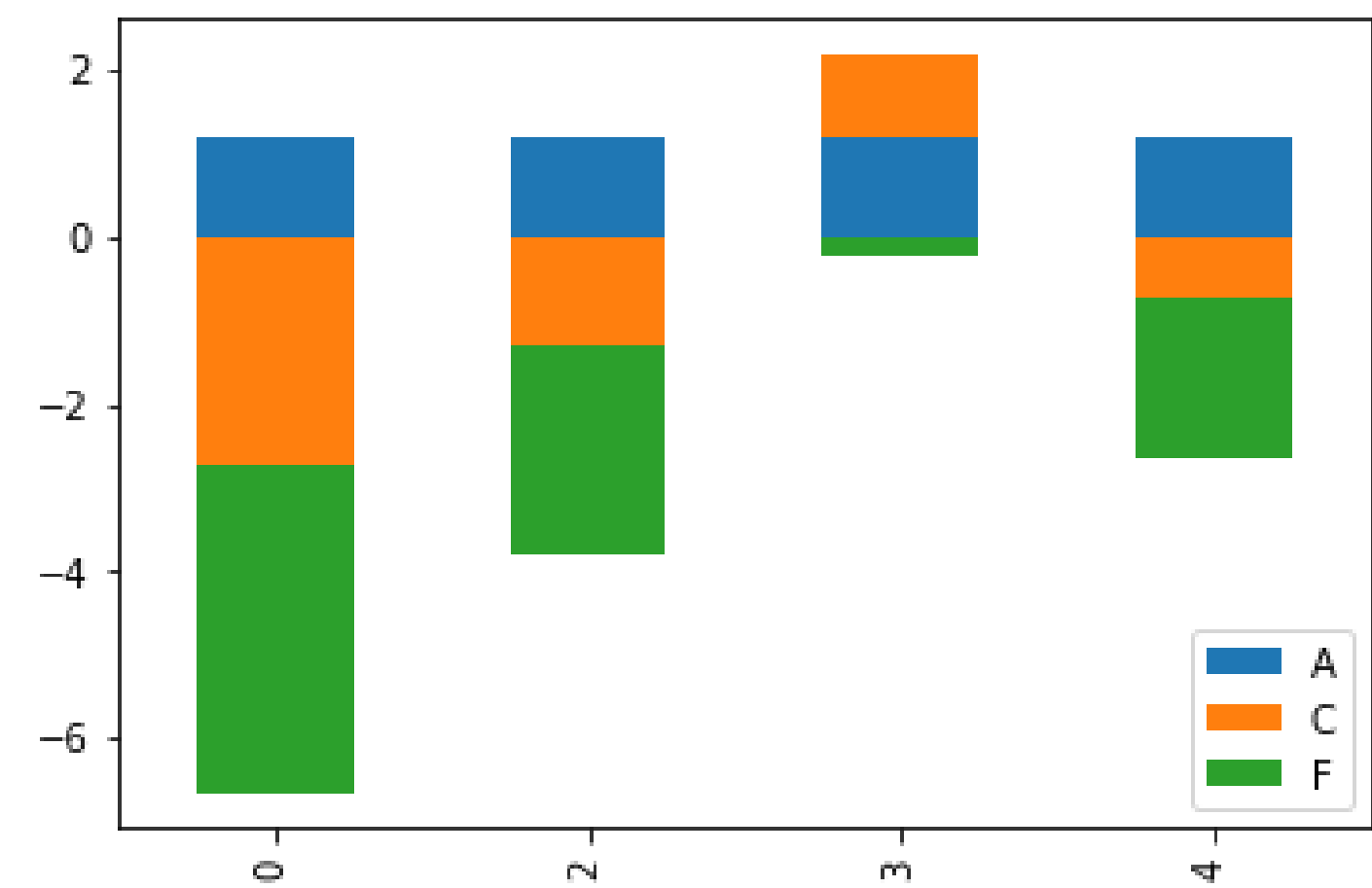
MORE PLOTTING WITH PANDAS

Some versatility

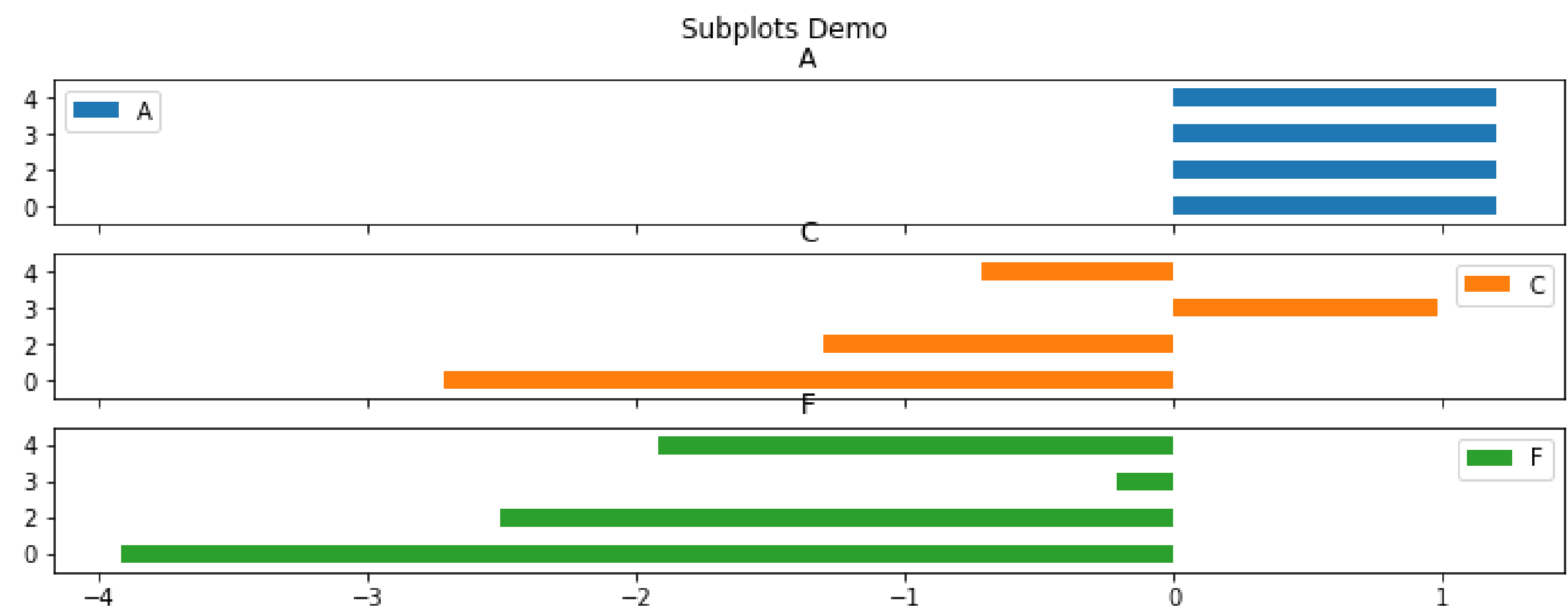
```
In [91]: df_demo[["A", "C", "F"]].plot(kind="bar", stacked=True);
```



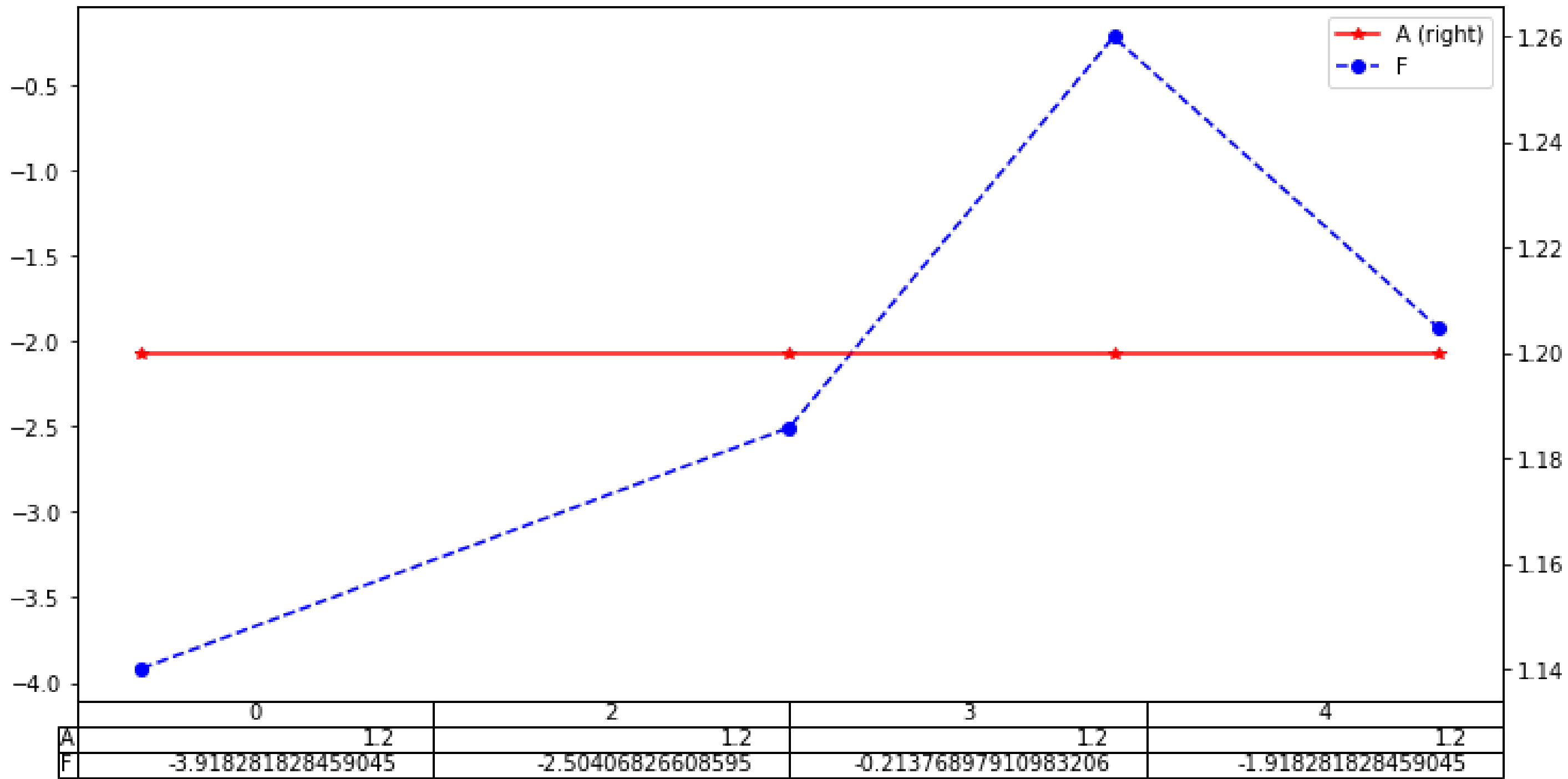
```
In [92]: df_demo[df_demo["F"] < 0][["A", "C", "F"]].plot(kind="bar", stacked=True);
```



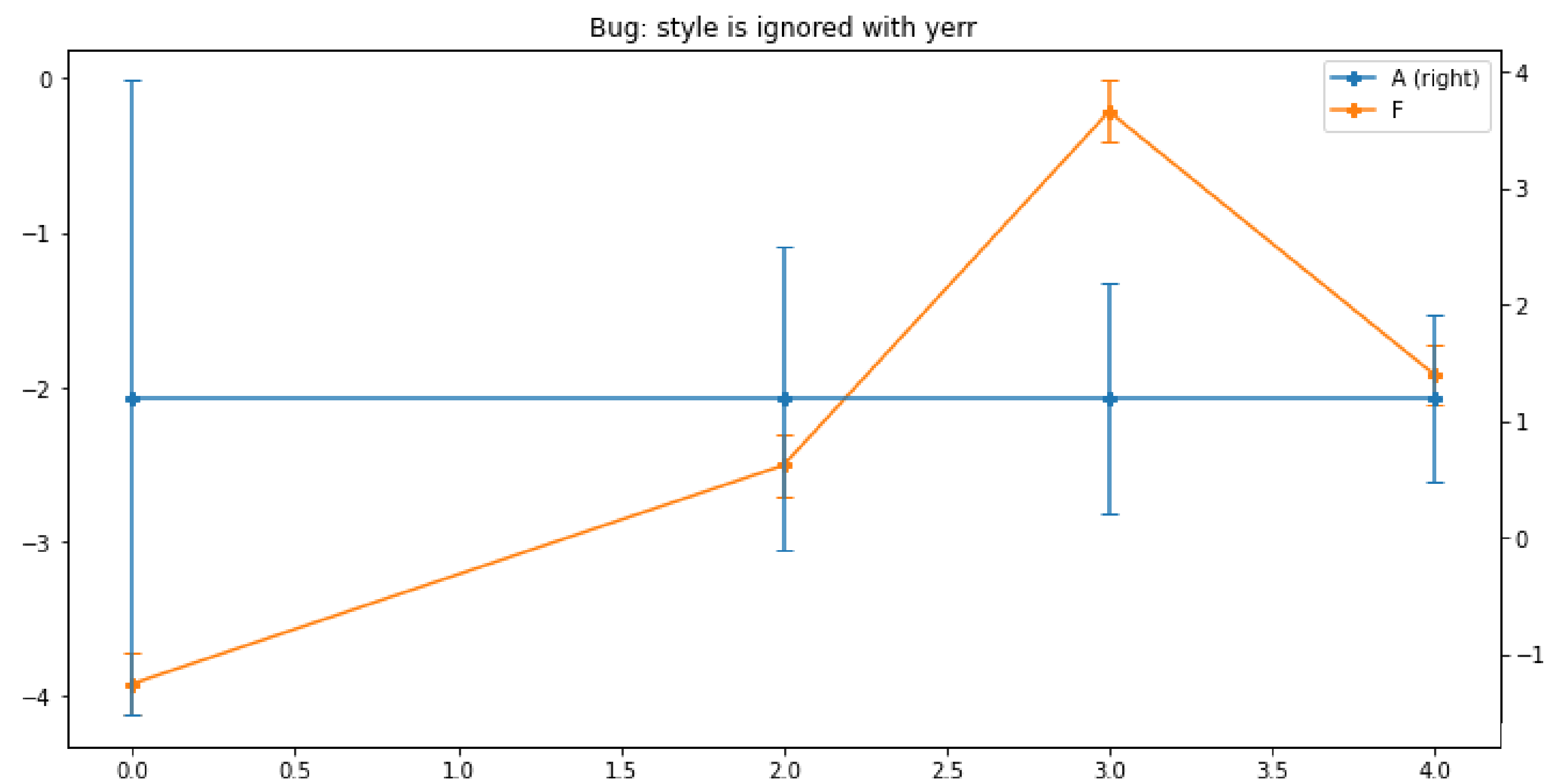
```
In [93]: df_demo[df_demo["F"] < 0][["A", "C", "F"]]\n        .plot(kind="barh", subplots=True, sharex=True, title="Subplots Demo", figsize=(12, 4));
```



```
In [94]: df_demo.loc[df_demo["F"] < 0, ["A", "F"]]\n        .plot(\n            style=["-*r", "--ob"],\n            secondary_y="A",\n            figsize=(12, 6),\n            table=True\n        );
```



```
In [95]: df_demo.loc[df_demo["F"] < 0, ["A", "F"]]\n        .plot(\n            style=["-*r", "--ob"],\n            secondary_y="A",\n            figsize=(12, 6),\n            yerr={\n                "A": df_demo[df_demo["F"] < 0]["C"],\n                "F": 0.2\n            },\n            capsize=4,\n            title="Bug: style is ignored with yerr",\n            marker="P"\n        );
```

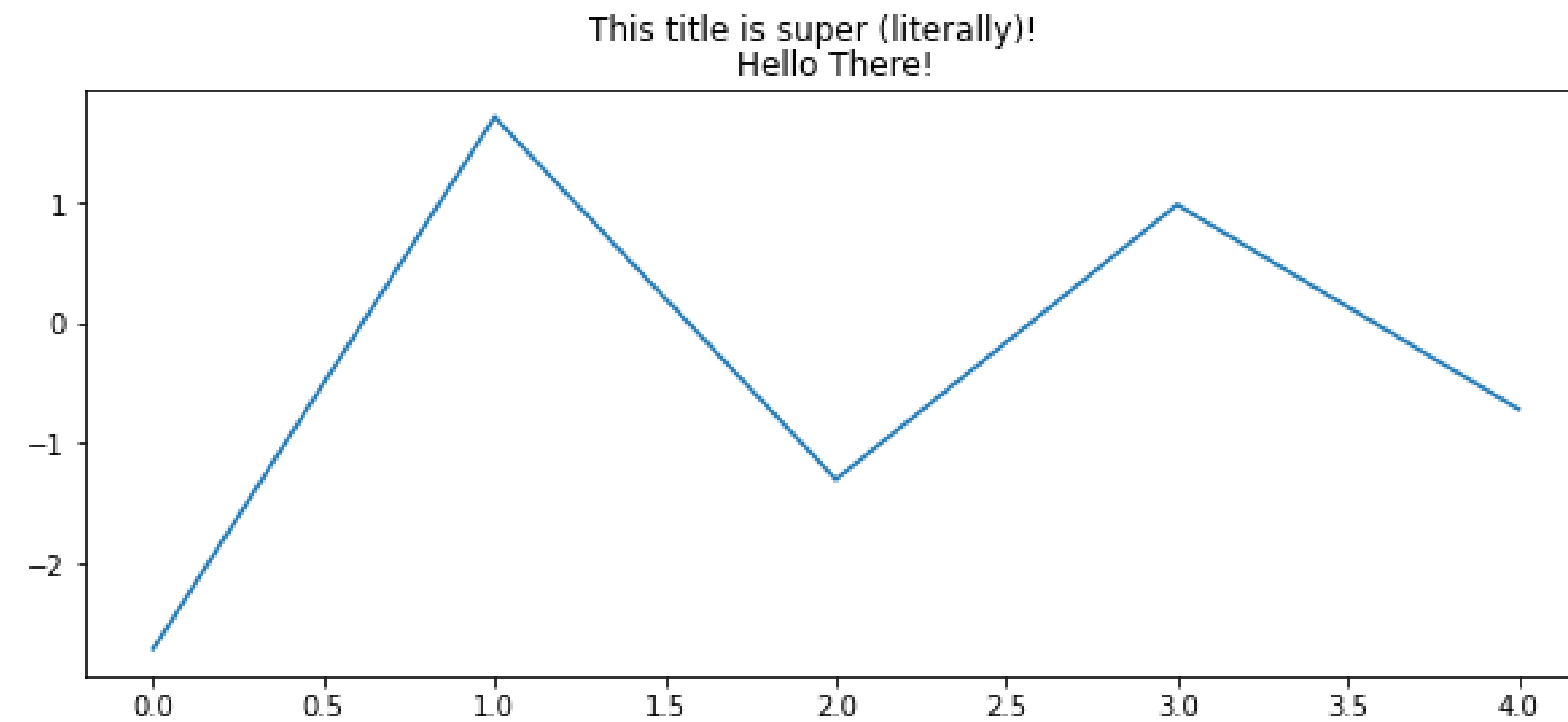


COMBINE PANDAS WITH MATPLOTLIB

- 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

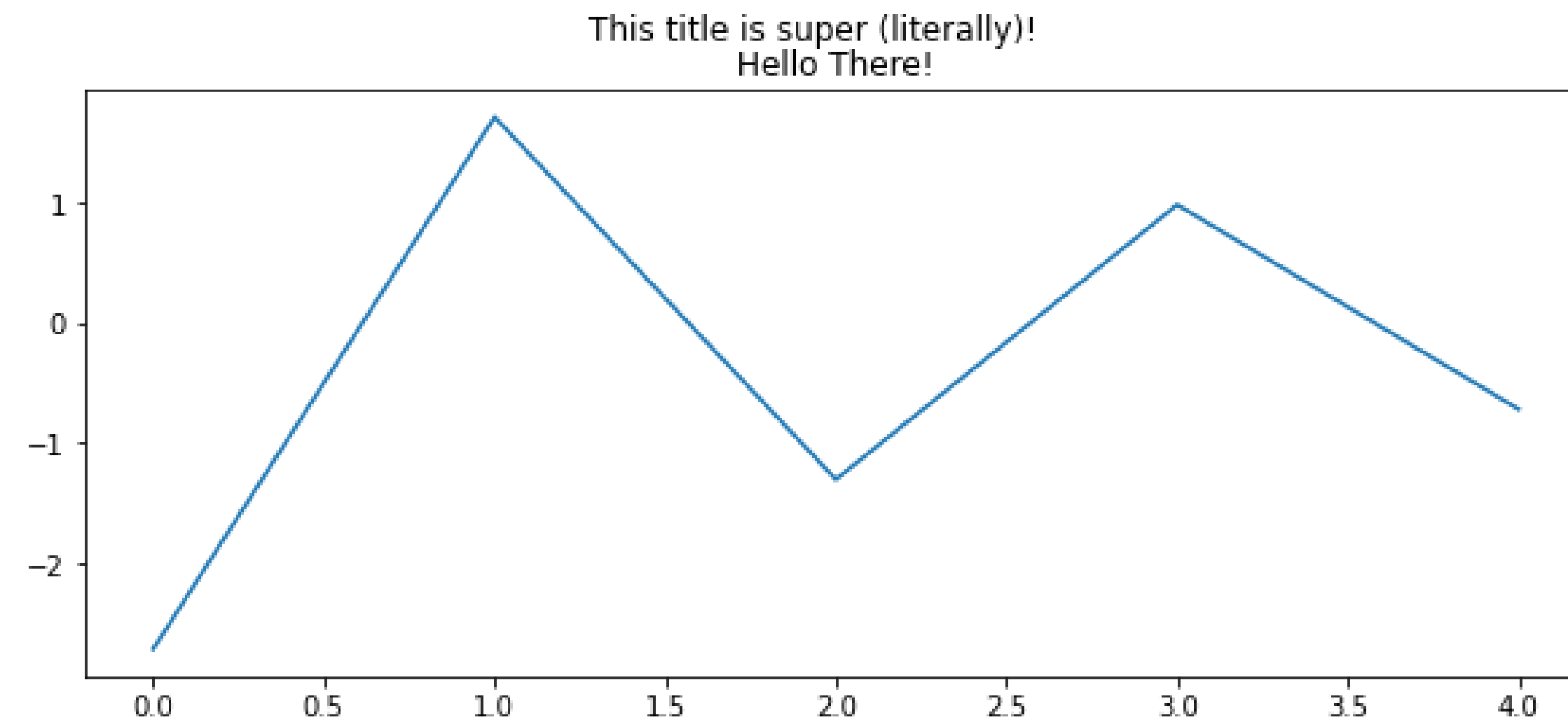
OPTION 1: PANDAS RETURNS AXIS

```
In [96]: ax = df_demo["C"].plot(figsize=(10, 4))
ax.set_title("Hello There!");
fig = ax.get_figure()
fig.suptitle("This title is super (literally)!");
```



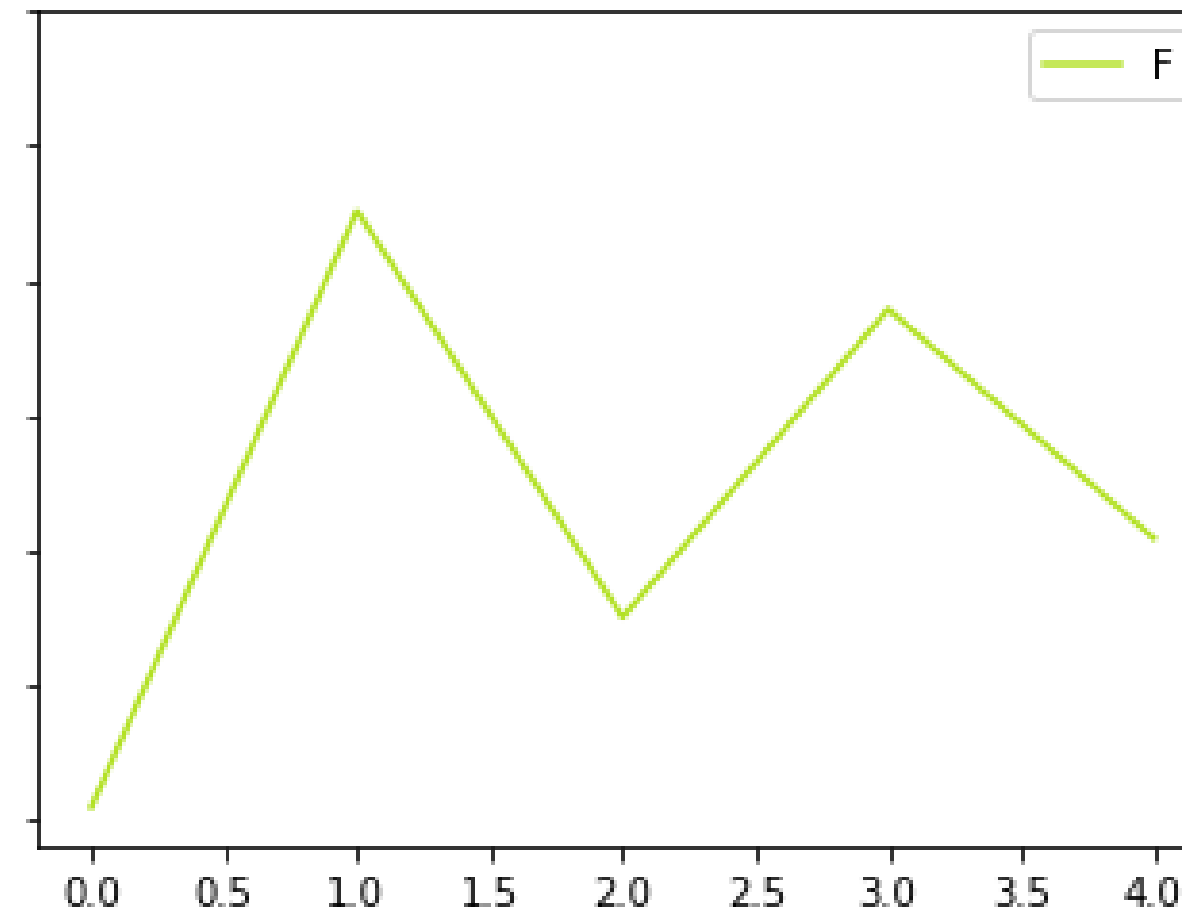
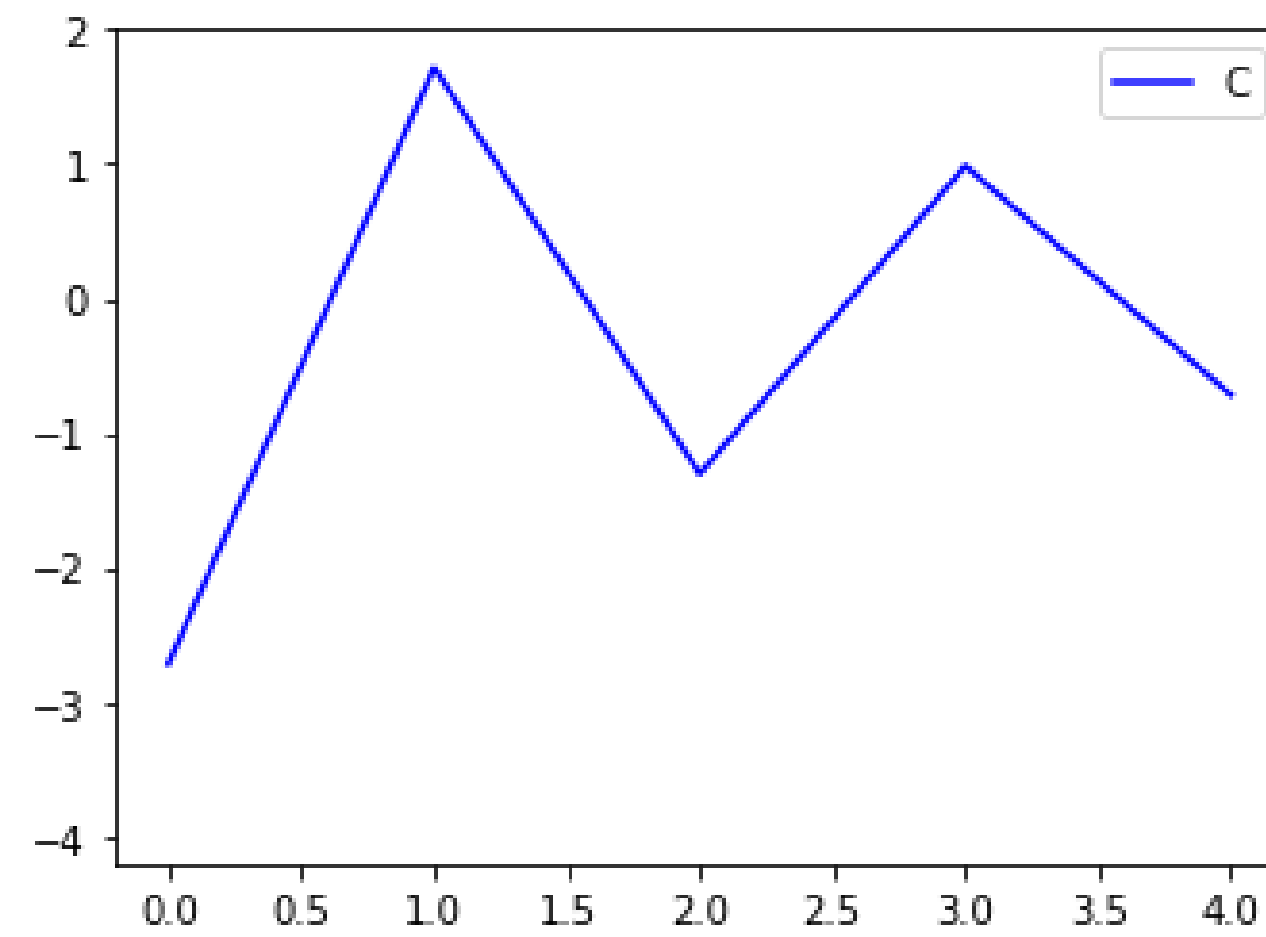
OPTION 2: DRAW ON MATPLOTLIB AXES

```
In [97]: fig, ax = plt.subplots(figsize=(10, 4))
df_demo["C"].plot(ax=ax)
ax.set_title("Hello There!");
fig.suptitle("This title is super (still, literally)!");
```



- We can also get fancy!

```
In [98]: 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)
```

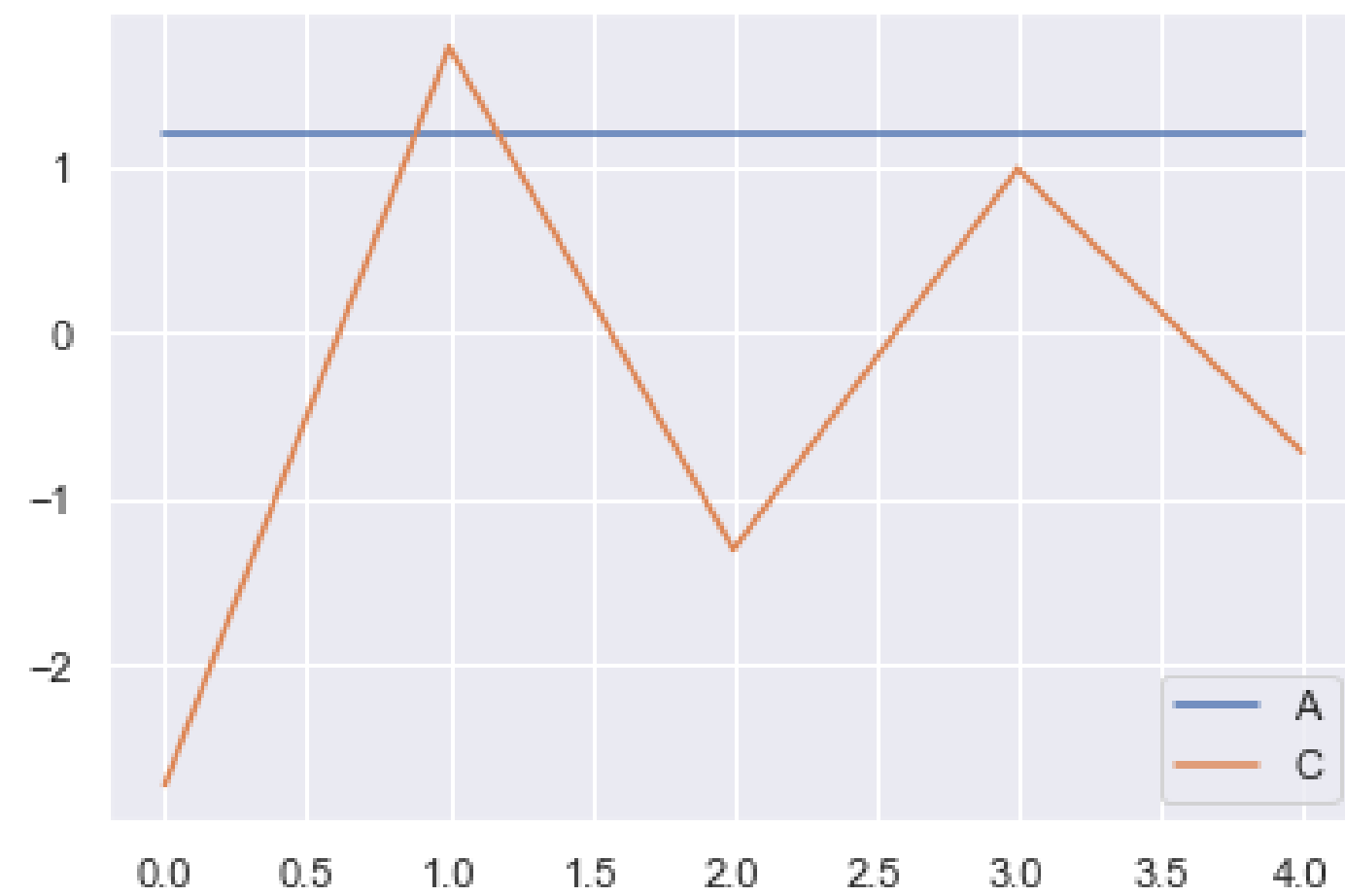


ASIDE: SEABORN

- 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/>

```
In [99]: import seaborn as sns  
sns.set() # set defaults
```

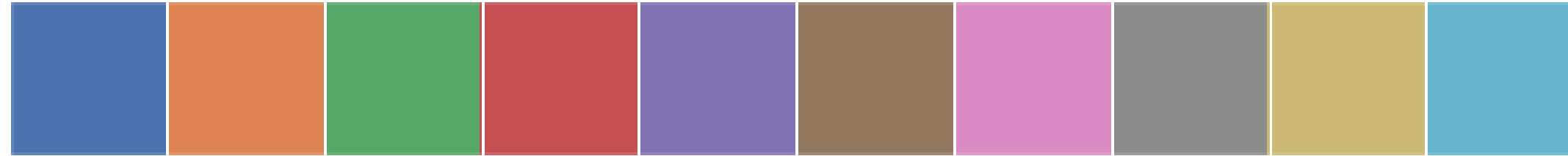
```
In [100]: df_demo[["A", "C"]].plot();
```



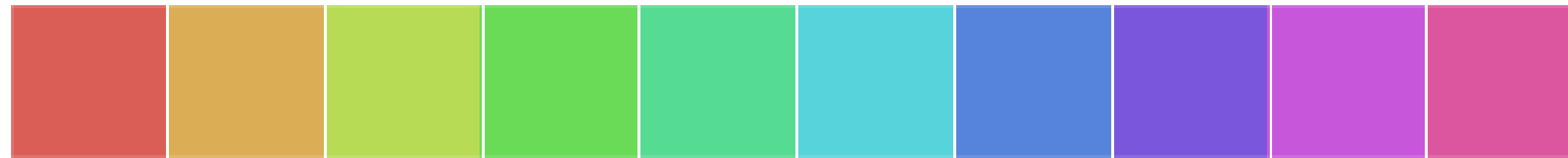
SEABORN COLOR PALETTE EXAMPLE

- [Documentation](#)

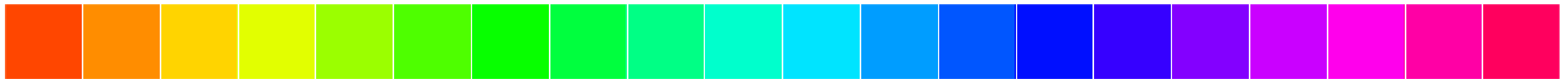
```
In [101]: sns.palplot(sns.color_palette())
```



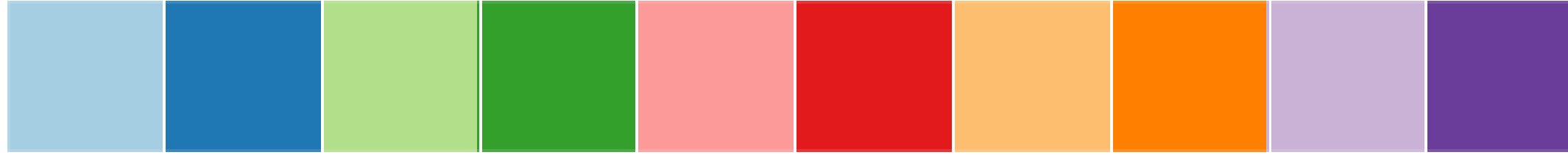
```
In [102]: sns.palplot(sns.color_palette("hls", 10))
```



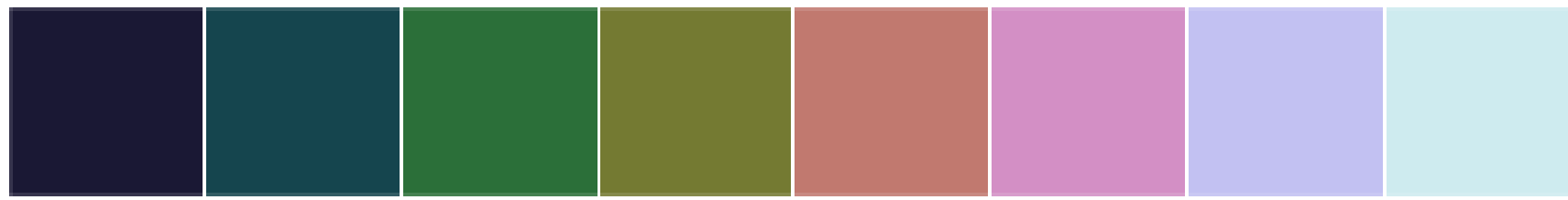
```
In [103]: sns.palplot(sns.color_palette("hsv", 20))
```



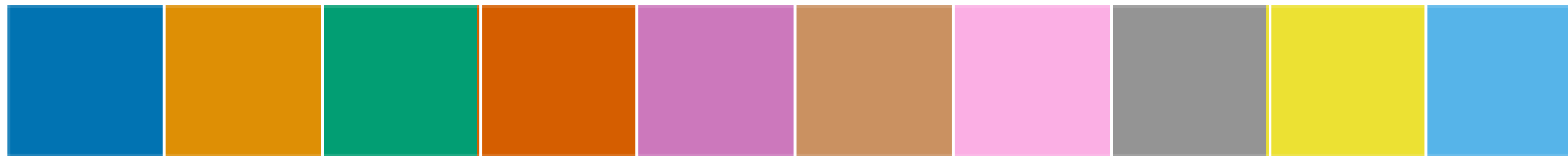
```
In [104]: sns.palplot(sns.color_palette("Paired", 10))
```



```
In [105]: sns.palplot(sns.color_palette("cubehelix", 8))
```



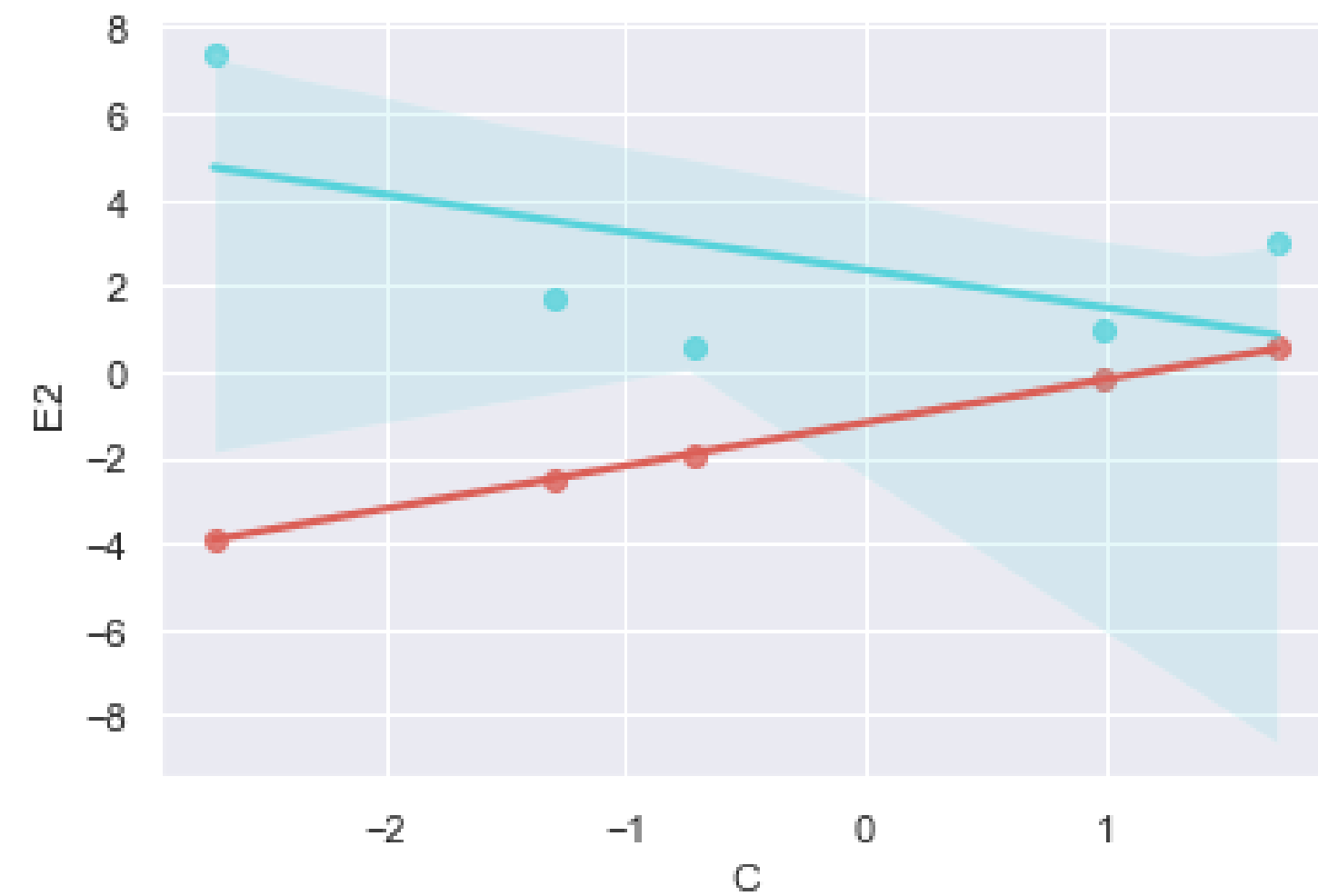
```
In [106]: sns.palplot(sns.color_palette("colorblind", 10))
```



SEABORN PLOT EXAMPLES

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

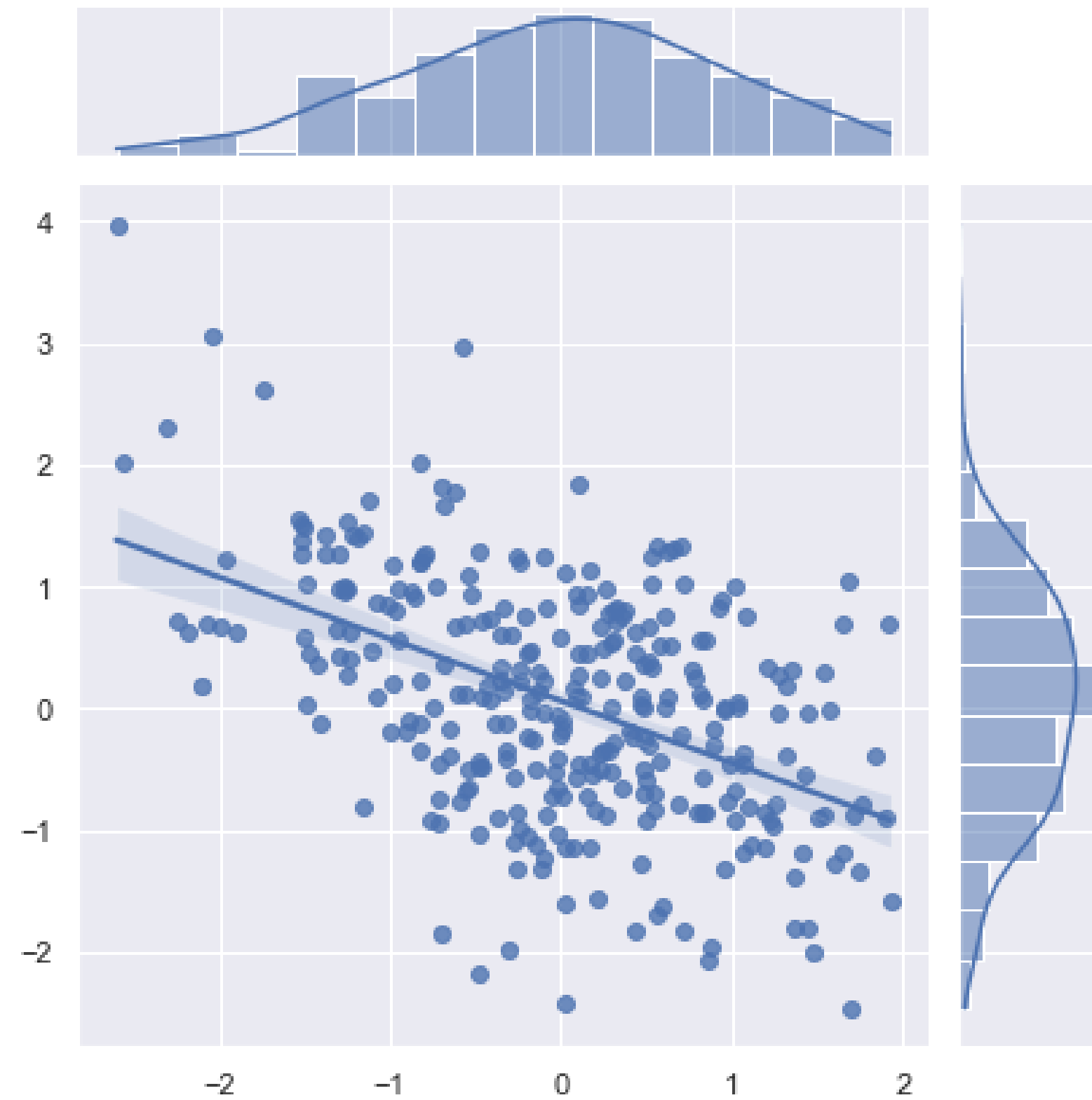
```
In [107]: with sns.color_palette("hls", 2):  
          sns.regplot(x="C", y="F", data=df_demo);  
          sns.regplot(x="C", y="E2", data=df_demo);
```



- 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

```
In [108]: x, y = np.random.multivariate_normal([0, 0], [[1, -.5], [-.5, 1]], size=300).T
```

```
In [109]: sns.jointplot(x=x, y=y, kind="reg");
```



- 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 threads
- Tell me when you're done with status icon in BigBlueButton: 👍

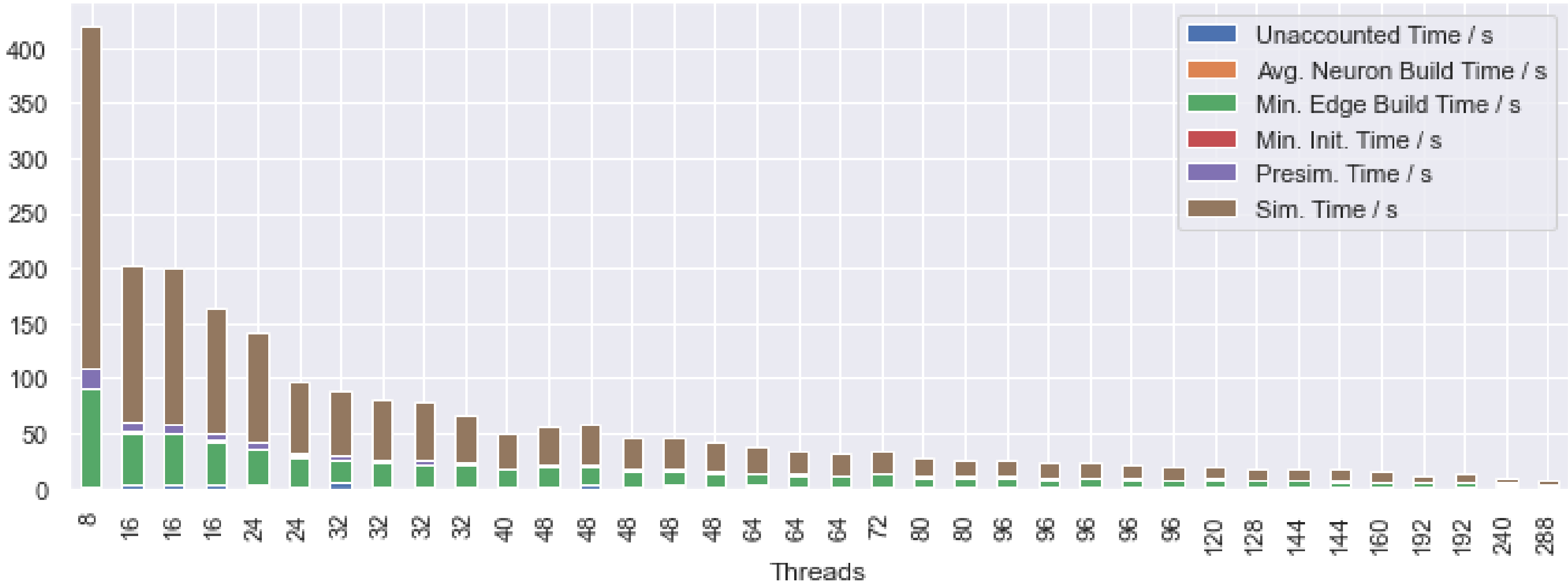
```
In [110]: cols = [  
    'Avg. Neuron Build Time / s',  
    'Min. Edge Build Time / s',  
    'Min. Init. Time / s',  
    'Presim. Time / s',  
    'Sim. Time / s'  
]  
df["Unaccounted Time / s"] = df['Runtime Program / s']  
for entry in cols:  
    df["Unaccounted Time / s"] = df["Unaccounted Time / s"] - df[entry]
```

```
In [111]: df[["Runtime Program / s", "Unaccounted Time / s", *cols]].head(2)
```

Out [111]:

	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
Threads							
8	420.42	2.09	0.29		88.12	1.14	17.26
16	202.15	2.43	0.28		47.98	0.70	7.95

```
In [112]: df[["Unaccounted Time / s", *cols]].plot(kind="bar", stacked=True, figsize=(12, 4));
```



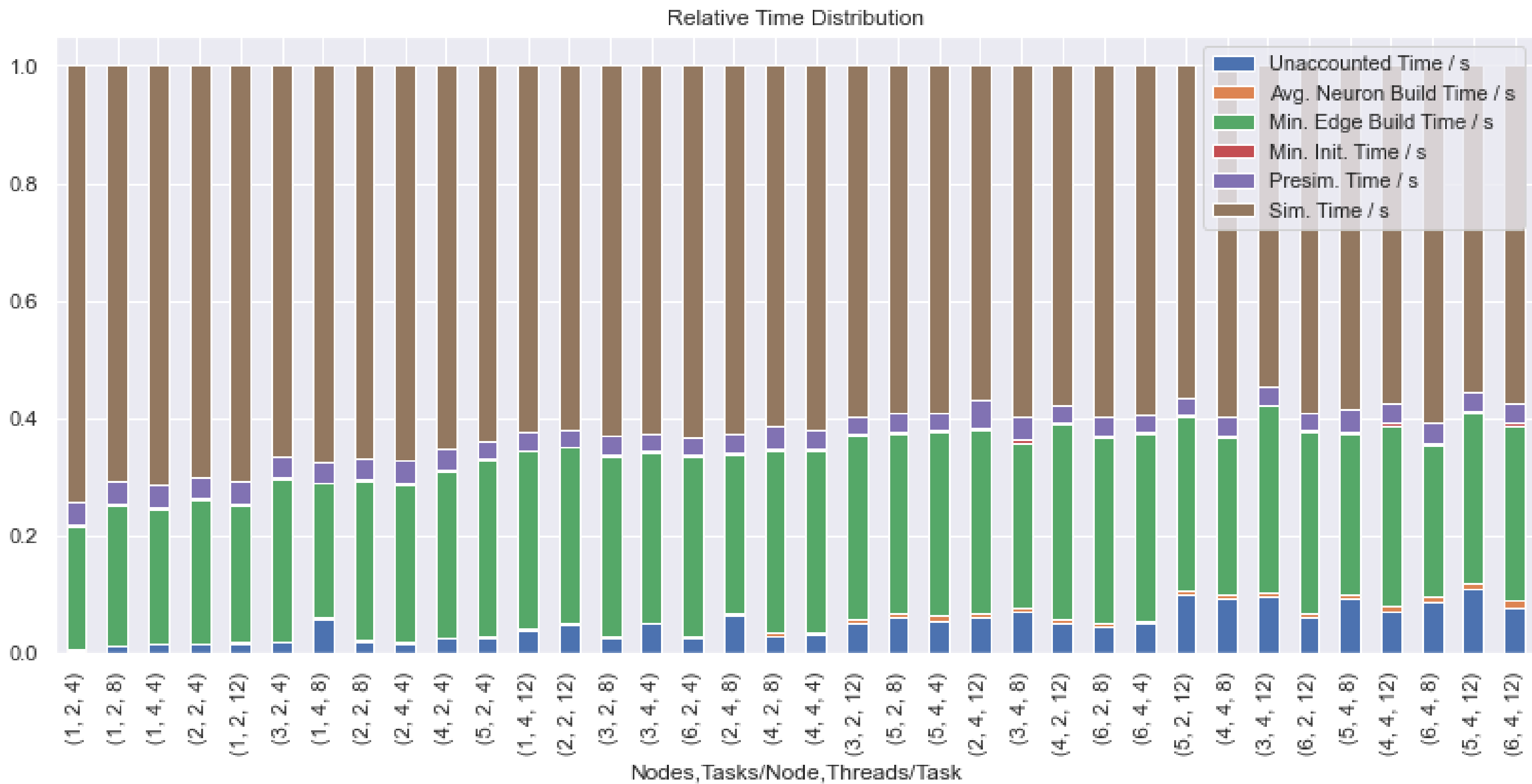
- Make it relative to the total program run time
- Slight complication: Our threads as indexes are not unique; we need to find new unique indexes
- Could be anything, but we use a multi index!

```
In [113]: df_multind = df.set_index(["Nodes", "Tasks/Node", "Threads/Task"])
df_multind.head()
```

Out [113]:

			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)	Number of Neurons	Nu Conr
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	112500	1265
		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	112500	1265
	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	112500	1265
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	112500	1265
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	112500	1265

```
In [114]: 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()` ("Split-apply-combine", [API](#), [User Guide](#))
 - Pivoting: `.pivot_table()` ([API](#), [User Guide](#)); also `.pivot()` (specialized version of `.pivot_table()`, [API](#))

```
In [115]: df.groupby("Nodes").mean()
```

Out [115]:

	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. Memor (Sum) / kl
Nodes														
1	5.333333	3.0	8.0	185.023333	10.0	1.0	0.220000	42.040000	42.838333	0.583333	...	7.226667	132.061667	4.806585e+0
2	5.333333	3.0	8.0	73.601667	10.0	1.0	0.168333	19.628333	20.313333	0.191667	...	2.725000	48.901667	4.975288e+0
3	5.333333	3.0	8.0	43.990000	10.0	1.0	0.138333	12.810000	13.305000	0.135000	...	1.426667	27.735000	5.511165e+0
4	5.333333	3.0	8.0	31.225000	10.0	1.0	0.116667	9.325000	9.740000	0.088333	...	1.066667	19.353333	5.325783e+0
5	5.333333	3.0	8.0	24.896667	10.0	1.0	0.140000	7.468333	7.790000	0.070000	...	0.771667	14.950000	6.075634e+0
6	5.333333	3.0	8.0	20.215000	10.0	1.0	0.106667	6.165000	6.406667	0.051667	...	0.630000	12.271667	6.060652e+0

6 rows × 21 columns

PIVOTING

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

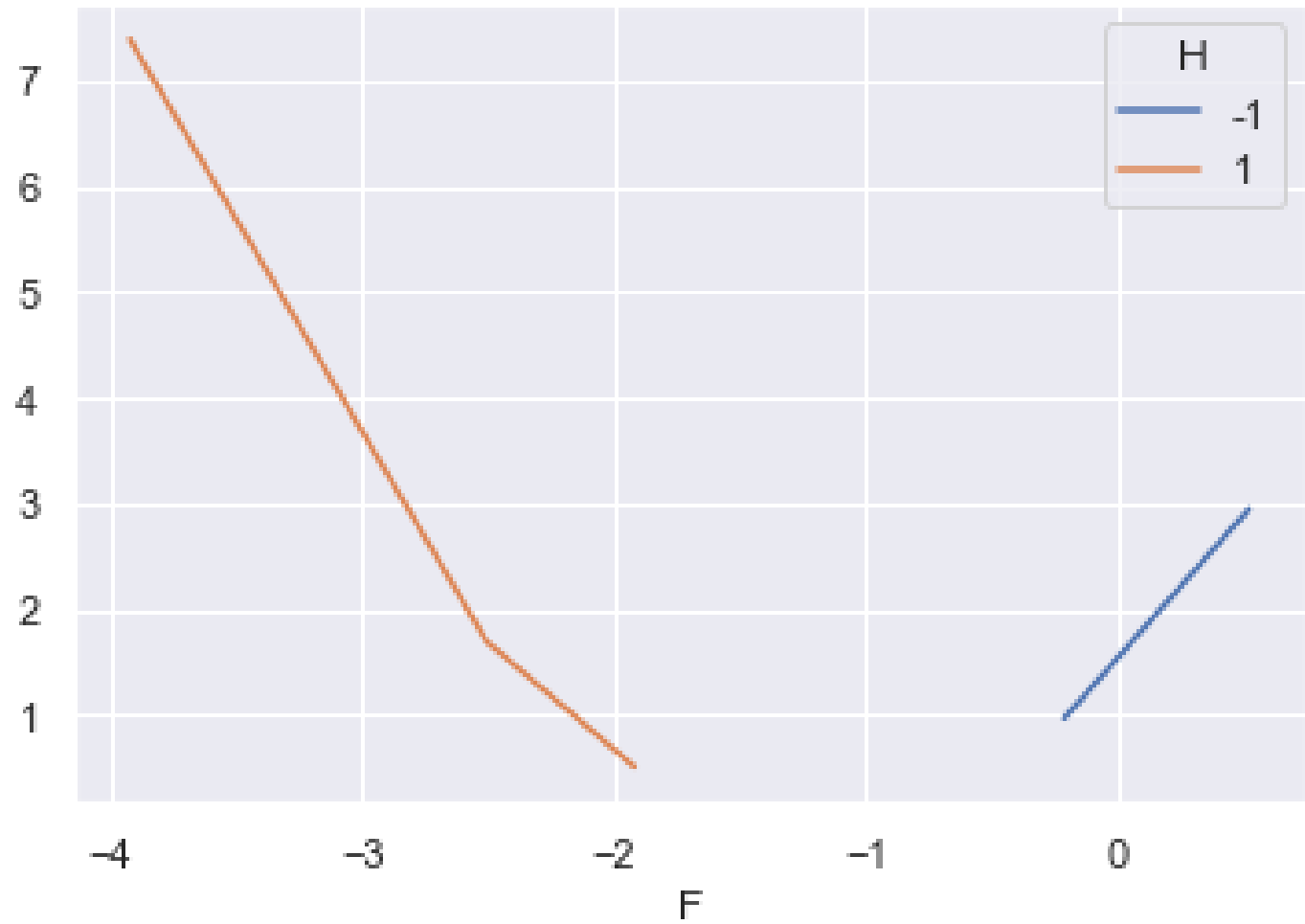
```
In [116]: df_demo["H"] = [(-1)**n for n in range(5)]
```

```
In [117]: df_pivot = df_demo.pivot_table(
    index="F",
    values="E2",
    columns="H"
)
df_pivot
```

Out[117]:

	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

```
In [118]: df_pivot.plot();
```

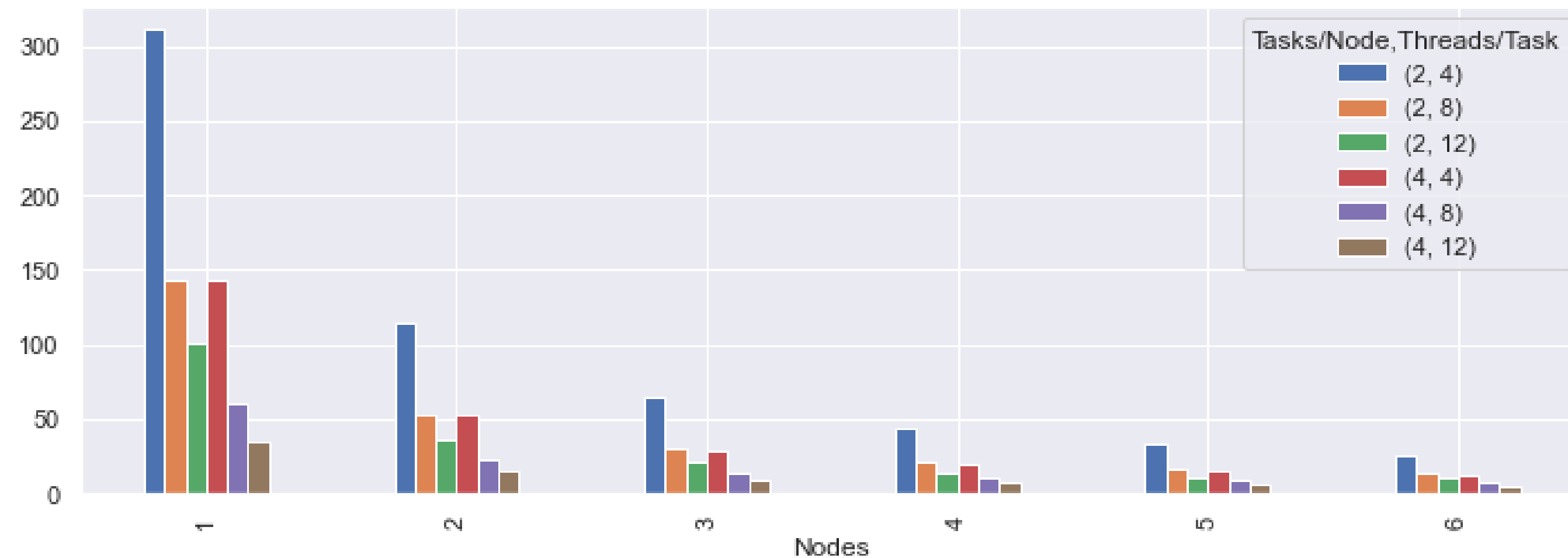


TASK 7

TASK

- 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 threads per task configurations
- Please plot a bar plot
- Tell me when you're done with status icon in BigBlueButton: 👍

```
In [119]: df.pivot_table(  
    index="Nodes",  
    columns=["Tasks/Node", "Threads/Task"],  
    values="Sim. Time / s",  
).plot(kind="bar", figsize=(12, 4));
```



TASK 7B (LIKE *BONUS*)

TASK

- Same pivot table as before (that is, `x` with nodes, and columns for Tasks/Node and Threads/Task)
- But now, use `Sim. Time / s` and `Presim. Time / s` as values to show
- Show them as a stack of those two values inside the pivot table
- Use Panda's functionality as much as possible!

Impossible?

- I gave up!
- Person who does this best / first: Personal certificate with my recommendation 😊

CONCLUSION

- Pandas works with and on data frames, which are central
- 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! 🥰

Feedback to 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](#)