DATA ANALYSIS AND PLOTTING IN PYTHON WITH PANDAS

Andreas Herten, Jülich Supercomputing Centre, Forschungszentrum Jülich, 4 September 2023

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
- Task 8
- Task 8B

COURSE SETUP

- 3½ hours, including break around 10:30
- Alternating between lecture and hands-on
- Please give status of hands-ons via 👍 as BigBlueButton status
- TAs and me in the room can help with issues, either in public chat or in 1:1 chat

- Please now open Jupyter Notebook of this session: https://go.fzj.de/jsc-pd
- 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
- Now at Pandas 2.0
- 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), pyarrow (Apache Arrow bindings in Python) …

FIRST STEPS

In [1]:
```
import pandas
```

In [2]:
```
import pandas as pd
```

In [3]:
```
pd.__version__
```

Out[3]:
```
'2.0.3'
```

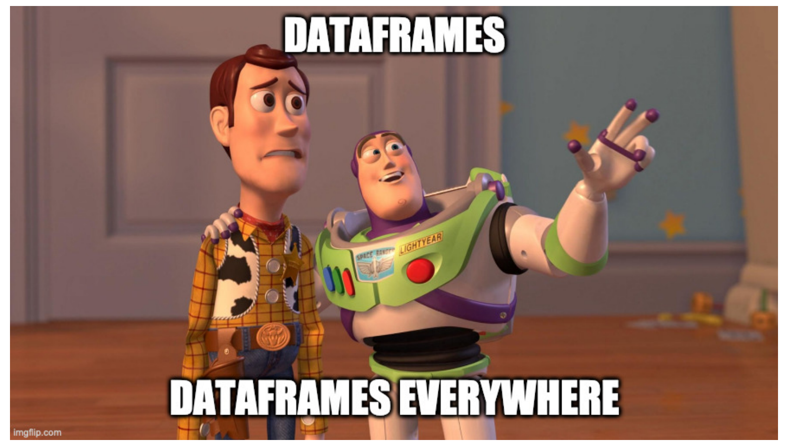
In [4]:
```
%pdoc pd
```

```
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-flo
        point data.
      - Size mutability: columns can be inserted and deleted from DataFram
        higher dimensional objects
      - Automatic and explicit data alignment: objects can be explicitly a
        to a set of labels, or the user can simply ignore the labels and l
        `Series`, `DataFrame`, etc. automatically align the data for you i
        computations.
      - Powerful, flexible group by functionality to perform split-apply-c
```

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, Parquet, SAS, SQL, Google BigQuery, STATA*

DATAFRAMES

Examples, finally

In [5]:
```
ages  = [41, 56, 56, 57, 39, 59, 43, 56, 38, 60]
```

In [6]:
```
pd.DataFrame(ages)
```

Out[6]:

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

In [7]:
```
df_ages = pd.DataFrame(ages)
df_ages.head(3)
```

Out[7]:

	0
0	41
1	56
2	56

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

```
In [8]: 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': [4
1, 56, 56, 57, 39, 59, 43, 56, 38, 60]}
```

```
In [9]: df_sample = pd.DataFrame(data)
        df_sample.head(4)
```

Out[9]:

	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 [10]: df_sample.columns
```

```
Out[10]: Index(['Name', 'Age'], dtype='object')
```

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

In [11]:
```
df_sample.index
```

Out[11]:
```
RangeIndex(start=0, stop=10, step=1)
```

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

In [12]:
```
df_sample.set_index("Name", inplace=True)
df_sample
```

Out[12]:

	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 [13]: df_sample.describe()
```

Out[13]:

	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 [14]: df_sample.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, Liu to Hall
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Age     10 non-null     int64
dtypes: int64(1)
memory usage: 160.0+ bytes
```

```
In [15]: df_sample.T
```

Out[15]:

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

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

```
Index(['Liu', 'Rowland', 'Rivers', 'Waters', 'Rice', 'Fields', 'Kerr',
       'Romero', 'Davis', 'Hall'],
```

- Also: Arithmetic operations

In [17]:
```
df_sample.multiply(2).head(3)
```

Out[17]:

	Age
Name	
Liu	82
Rowland	112
Rivers	112

In [18]:
```
df_sample.reset_index().multiply(2).head(3)
```

Out[18]:

	Name	Age
0	LiuLiu	82
1	RowlandRowland	112
2	RiversRivers	112

In [19]:
```
(df_sample / 2).head(3)
```

Out[19]:

	Age
Name	
Liu	20.5
Rowland	28.0
Rivers	28.0

In [20]:
```
(df_sample * df_sample).head(3)
```

Out[20]:

	Age
Name	
Liu	1681
Rowland	3136
Rivers	3136

In [21]:
```
def mysquare(number: float) -> float:
    return number*number

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

Out[21]:

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

In [23]:
```
df_sample.apply(np.square).head()
```

Out[23]:

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

Logical operations allowed as well

In [24]:
```
df_sample > 40
```

Out[24]:

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

In [25]:
```
df_sample.apply(mysquare).head() == df_sample.apply(lambda x: x*x).head()
```

Out[25]:

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

TASK 1

TASK

- 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 [26]:

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

In [27]:

```
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[27]:

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 [28]:

```
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[28]:

	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 [29]:

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

Out[29]:

	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 [30]:
```python
df_demo.round(2).tail(2)
```

Out[30]:

	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 [31]:
```python
df_demo.round(2)[["A", "C"]].sum()
```

Out[31]:
```
A     6.00
C    -2.03
dtype: float64
```

In [32]:
```python
print(df_demo.round(2).to_latex())
```

```
\begin{tabular}{lrlrll}
\toprule
 & A & B & C & D & E \\
\midrule
0 & 1.200000 & 2018-02-26 00:00:00 & -2.720000 & This & Same \\
1 & 1.200000 & 2018-02-26 00:00:00 & 1.720000 & column & Same \\
2 & 1.200000 & 2018-02-26 00:00:00 & -1.300000 & has & Same \\
3 & 1.200000 & 2018-02-26 00:00:00 & 0.990000 & entries & Same \\
4 & 1.200000 & 2018-02-26 00:00:00 & -0.720000 & 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]
}
```

In [33]:
```
pd.read_json("data-lost.json").set_index("Character").sort_index()
```

Out[33]:

	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

TASK

- 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 [34]:
```
!head data-nest.csv
```

```
id,Nodes,Tasks/Node,Threads/Task,Runtime Program / s,Scale,Plastic,Avg. Neuron Build Time / s,Min. Edge Build Ti
me / s,Max. Edge Build Time / s,Min. Init. Time / s,Max. Init. Time / s,Presim. Time / s,Sim. Time / s,Virt. Mem
ory (Sum) / kB,Local Spike Counter (Sum),Average Rate (Sum),Number of Neurons,Number of Connections,Min. Delay,M
ax. 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 [35]:
```
df = pd.read_csv("data-nest.csv")
df.head()
```

Out[35]:

	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
0	5	1	2	4	420.42	10	True	0.29	88.12	88.18	...	1.20
1	5	1	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=_NoDefault.no_default, delimiter=None, header='infer', names=_NoDefault.no_default, index_col=None,
usecols=None, 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=None,
infer_datetime_format=_NoDefault.no_default, keep_date_col=False, date_parser=_NoDefault.no_default, date_format=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, encoding_errors='strict', dialect=None, on_bad_lines='error',
delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None, storage_options=None, dtype_backend=_NoDefault.no_default)
```

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 [36]:
```
df_demo.head(3)
```

Out[36]:

	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 [37]:
```
df_demo['C']
```

Out[37]:
```
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 [38]:
```
df_demo.C
```

Out[38]:
```
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 [39]:
```
my_slice = ['A', 'C']
df_demo[my_slice]
```

Out[39]:

	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 [40]:
```
df_demo[1:3]
```

Out[40]:

	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 [41]:
```
df_demo[1:6:2]
```

Out[41]:

	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 [42]:
```
df_demo[1:3]
```

Out[42]:

	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 [43]:
```
df_demo.sort_values("C")[1:3]
```

Out[43]:

	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 [44]:
```
df_demo.iloc[1:3]
```

Out[44]:

	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 [45]:
```
df_demo.iloc[1:3, [0, 2]]
```

Out[45]:

	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 [46]:
```
df_demo_indexed = df_demo.set_index("D")
df_demo_indexed
```

Out[46]:

D	A	B	C	E
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 [47]:
```
df_demo_indexed.loc["entries"]
```

Out[47]:

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

In [48]:
```
df_demo_indexed.loc[["has", "entries"], ["A", "C"]]
```

Out[48]:

D	A	C
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 [49]:
```
df_demo[df_demo["C"] > 0]
```

Out[49]:

	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 [50]:
```
df_demo["C"] > 0
```

Out[50]:
```
0    False
1     True
2    False
3     True
4    False
Name: C, dtype: bool
```

In [51]:
```
df_demo[(df_demo["C"] < 0) & (df_demo["D"] == "entries")]
```

Out[51]:

	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()`
- 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 [52]:

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

Out[52]:

	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 [53]:

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

Out[53]:

	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 [54]:
```
df_demo.insert(df_demo.shape[1] - 1, "E2", df_demo["C"] ** 2)
df_demo.head(3)
```

Out[54]:

	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 [55]:
```
df_demo.tail(3)
```

Out[55]:

	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

COMBINING FRAMES

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

In [56]:
```
df_1 = pd.DataFrame({"Key": ["First", "Second"], "Value": [1, 1]})
df_1
```

Out[56]:

	Key	Value
0	First	1
1	Second	1

In [57]:
```
df_2 = pd.DataFrame({"Key": ["First", "Second"], "Value": [2, 2]})
df_2
```

Out[57]:

	Key	Value
0	First	2
1	Second	2

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

```
In [58]: pd.concat([df_1, df_2])
```

Out[58]:

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

- Same, but re-index

```
In [59]: pd.concat([df_1, df_2], ignore_index=True)
```

Out[59]:

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

- Concat, but horizontally

In [60]:
```
pd.concat([df_1, df_2], axis=1)
```

Out[60]:

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

- Merge on common column

In [61]:
```
pd.merge(df_1, df_2, on="Key")
```

Out[61]:

	Key	Value_x	Value_y
0	First	1	2
1	Second	1	2

`.concat()` can also be used to append rows to a DataFrame:

In [62]:
```
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
)
```

Out[62]:

	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

TASK 3

TASK

- 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 [63]:
```
df["Threads"] = df["Nodes"] * df["Tasks/Node"] * df["Threads/Task"]
df.head()
```

Out[63]:

	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	1265738500
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	1265738500
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	1265738500
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	1265738500
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	1265738500

5 rows × 22 columns

In [64]:
```
df.columns
```

Out[64]:
```
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 (
       'Average Rate (Sum)', 'Number of Neurons', 'Number of Connections'
       'Min. Delay', 'Max. Delay', 'Threads'],
      dtype='object')
```

ASIDE: PLOTTING WITHOUT PANDAS

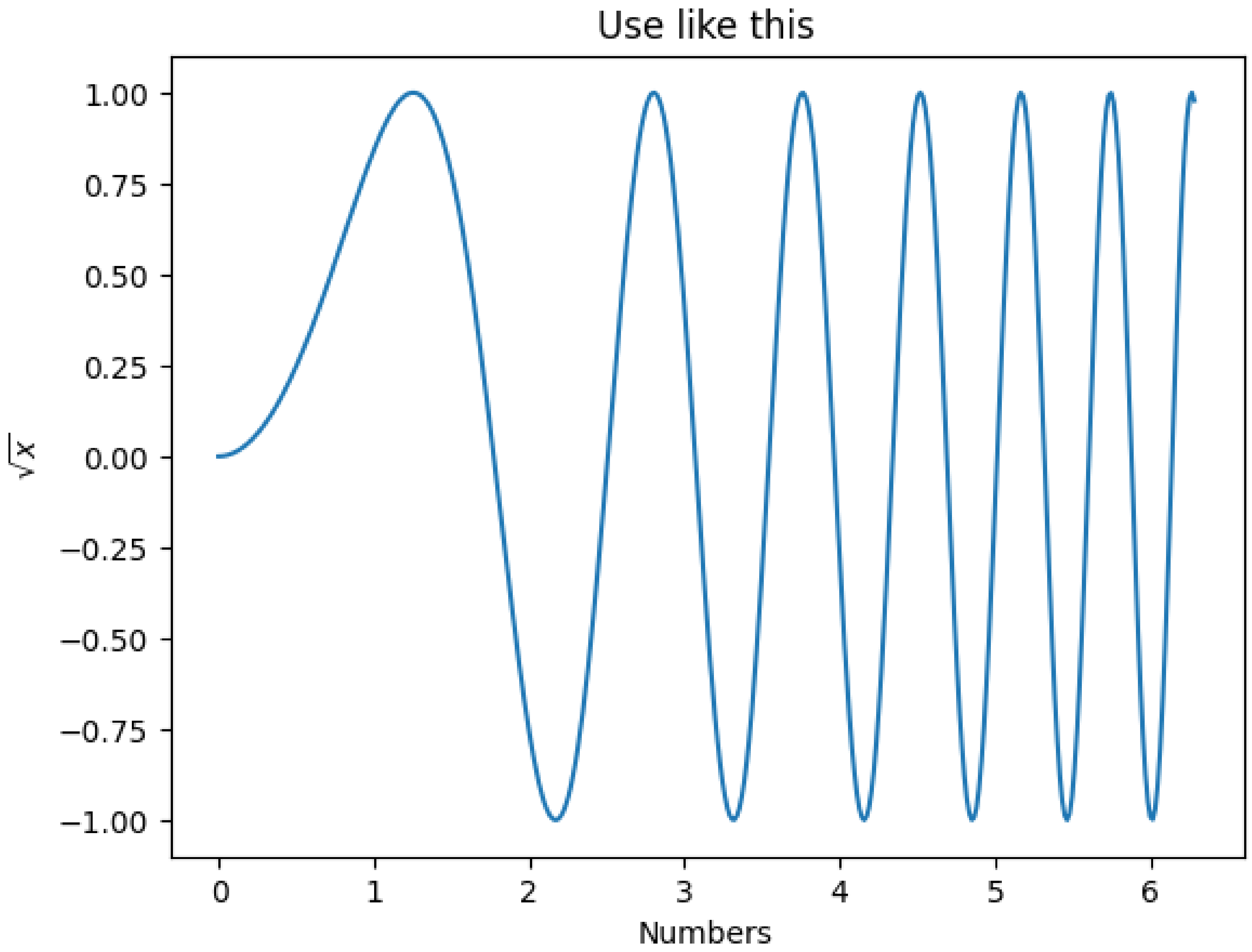
Matplotlib 101

- 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 [65]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [66]:
```
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)
```

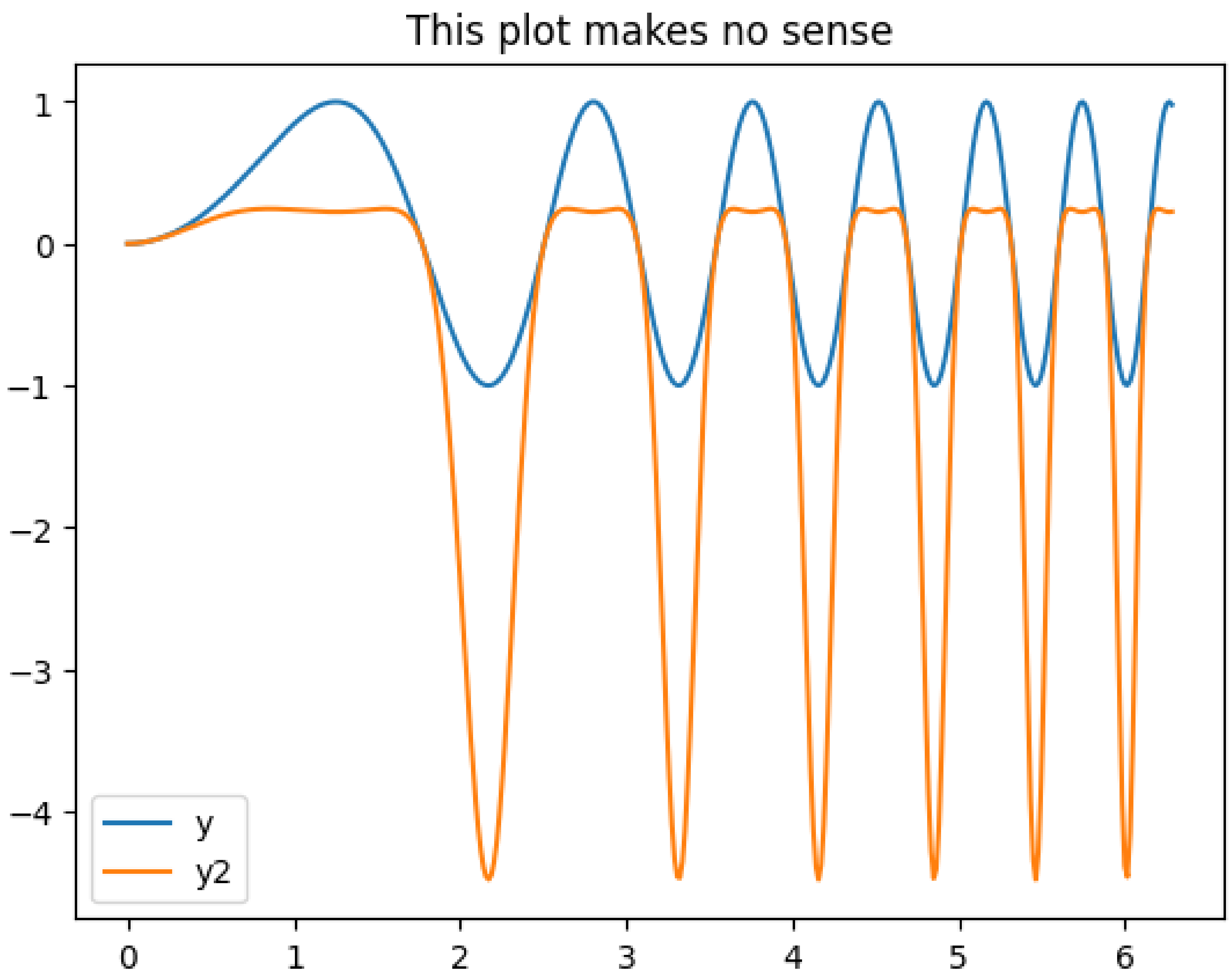
In [67]:
```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Use like this')
ax.set_xlabel("Numbers");
ax.set_ylabel("$\sqrt{x}$");
```



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

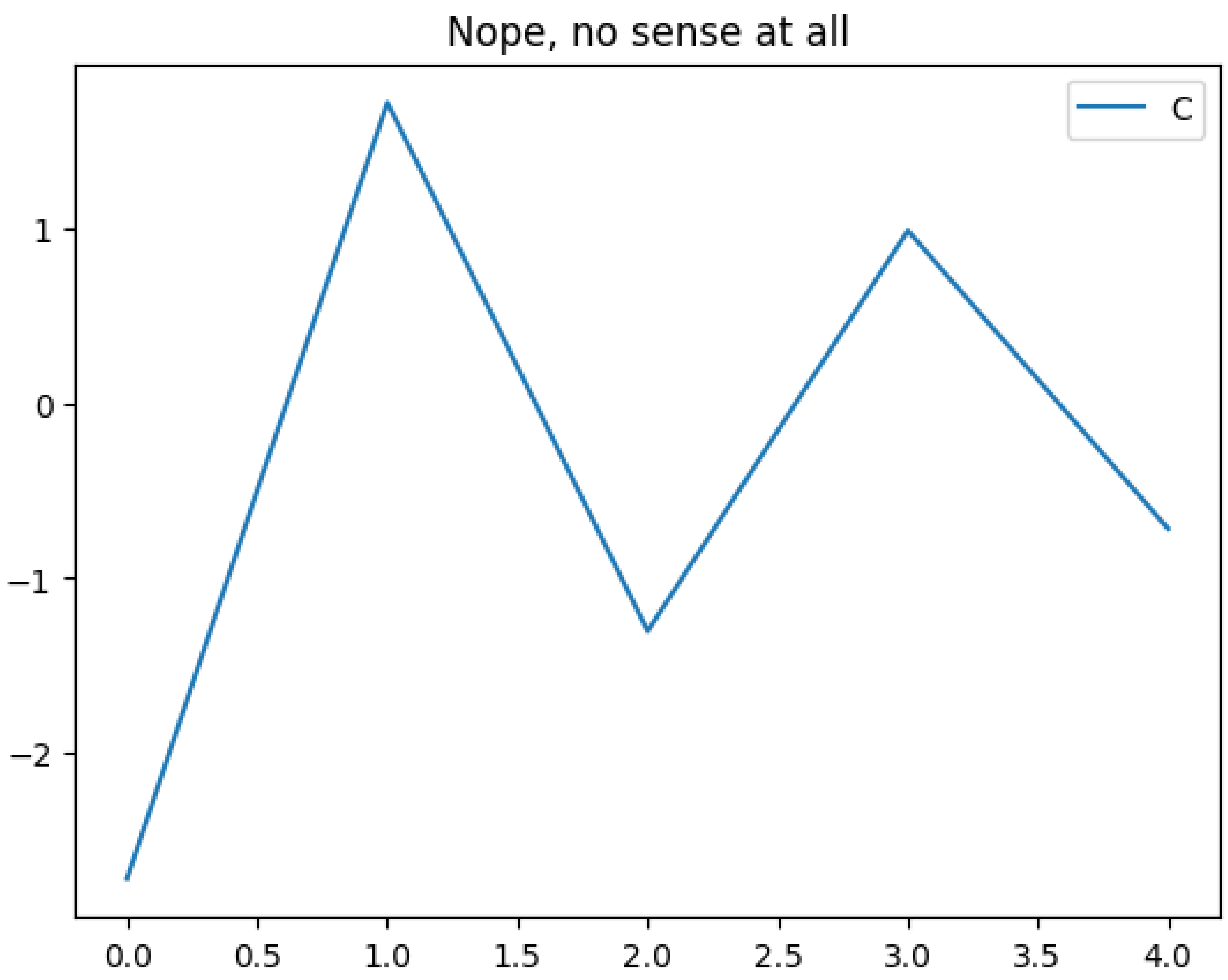
In [68]:
```
y2 = y/np.exp(y*1.5)
```

In [69]:
```
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 [70]:
```
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

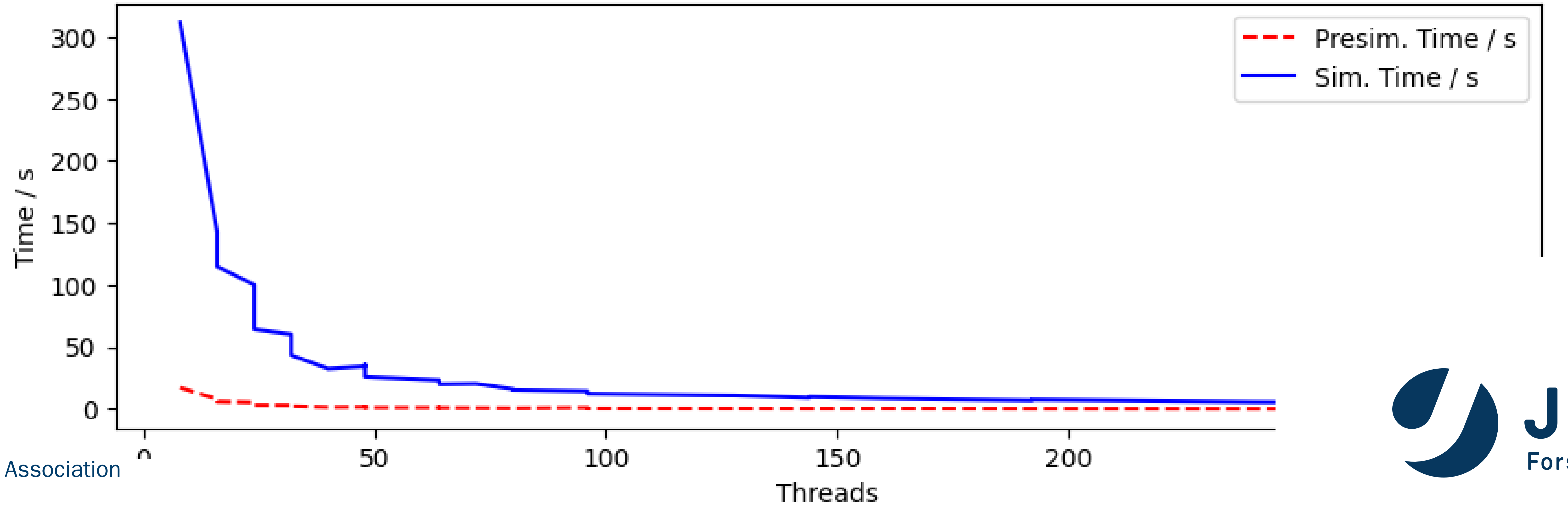
- Sort the Nest data frame by threads
- Plot `"Presim. Time / s"` and `"Sim. Time / s"` of our data frame `df` as a function of threads
- Use a dashed, red line for `"Presim. Time / s"`, a blue line for `"Sim. Time / s"` (see API description)
- Don't forget to label your axes and to add a legend *(1st rule of plotting)*
- Tell me when you're done with status icon in BigBlueButton: 👍

In [71]:

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

In [72]:

```
fig, ax = plt.subplots(figsize=(10, 3))
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');
```

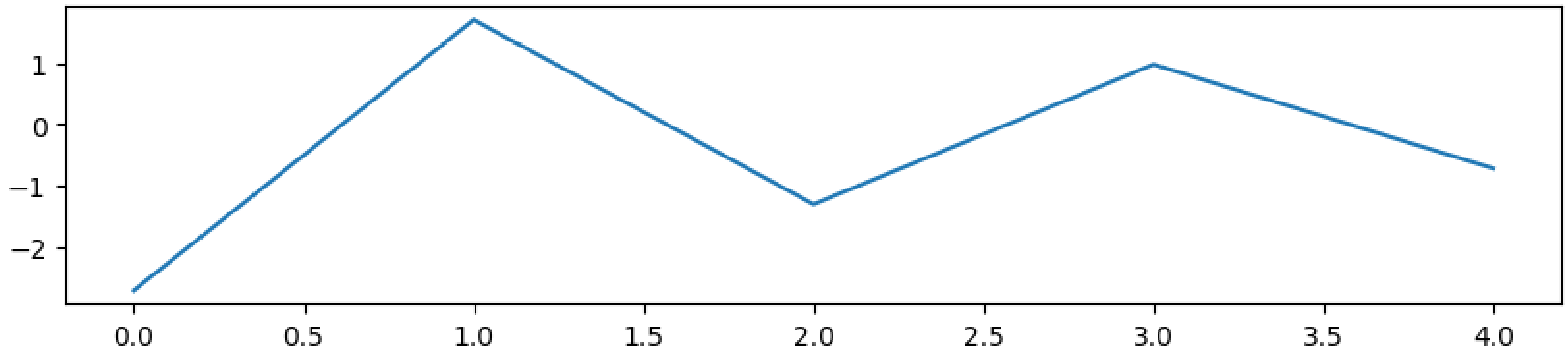


PLOTTING WITH PANDAS

- 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
  - `**kwds`: Non-parsed keyword passed to Matplotlib's plotting methoc

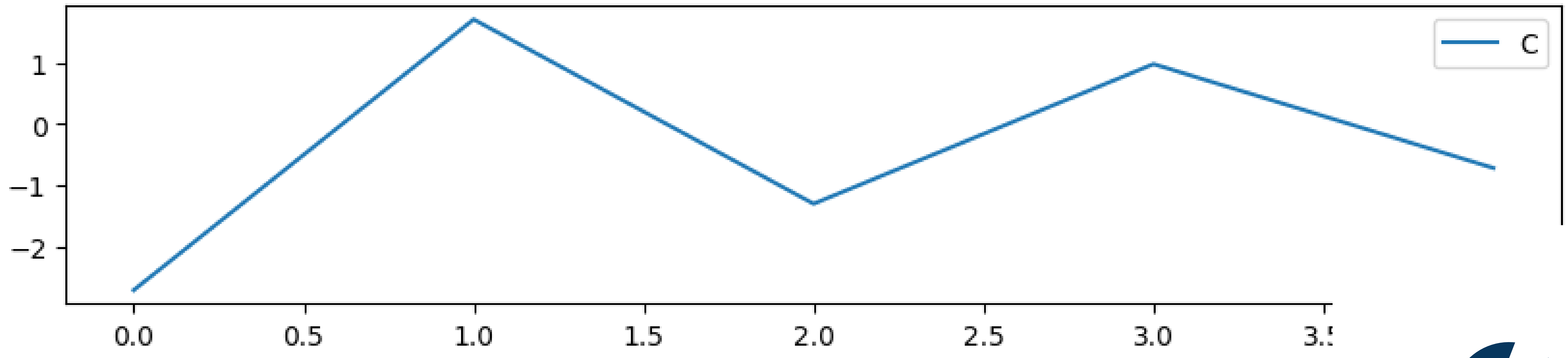
- Either slice and plot…

```
In [73]: df_demo["C"].plot(figsize=(10, 2));
```



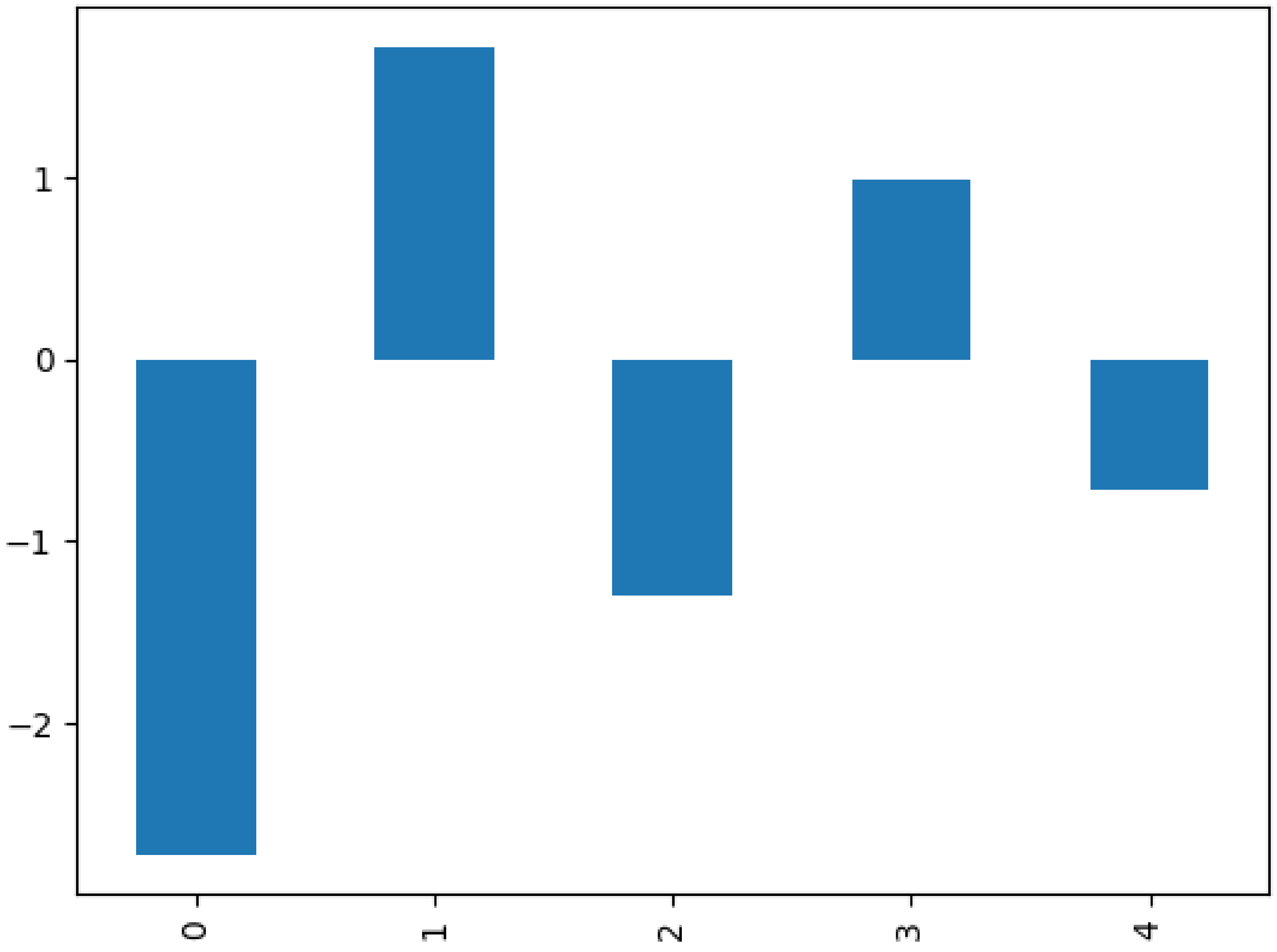
- … or plot and select

```
In [74]: df_demo.plot(y="C", figsize=(10, 2));
```



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

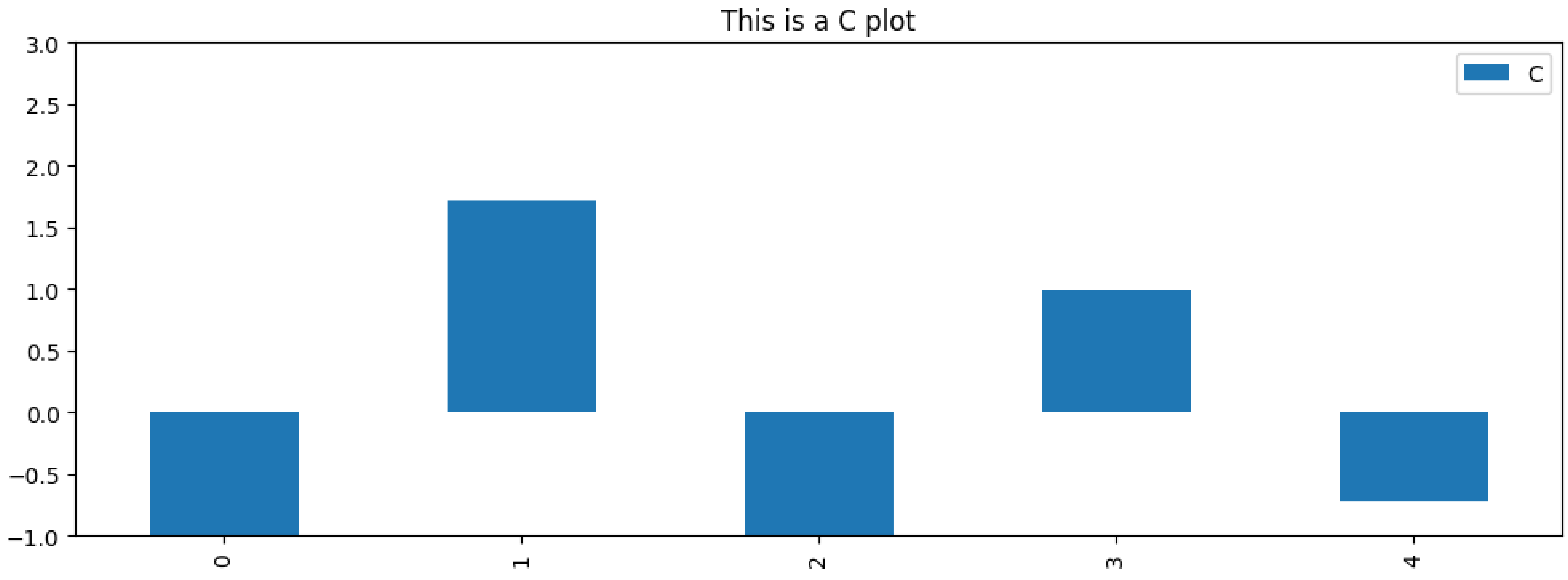
In [75]:
```
df_demo["C"].plot(kind="bar");
```



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

In [76]:
```
df_demo["C"].plot.bar();
```

In [77]:
```
df_demo["C"].plot(kind="bar", legend=True, figsize=(12, 4), ylim=(-1, 3), title="This is a C plot");
```



TASK 5

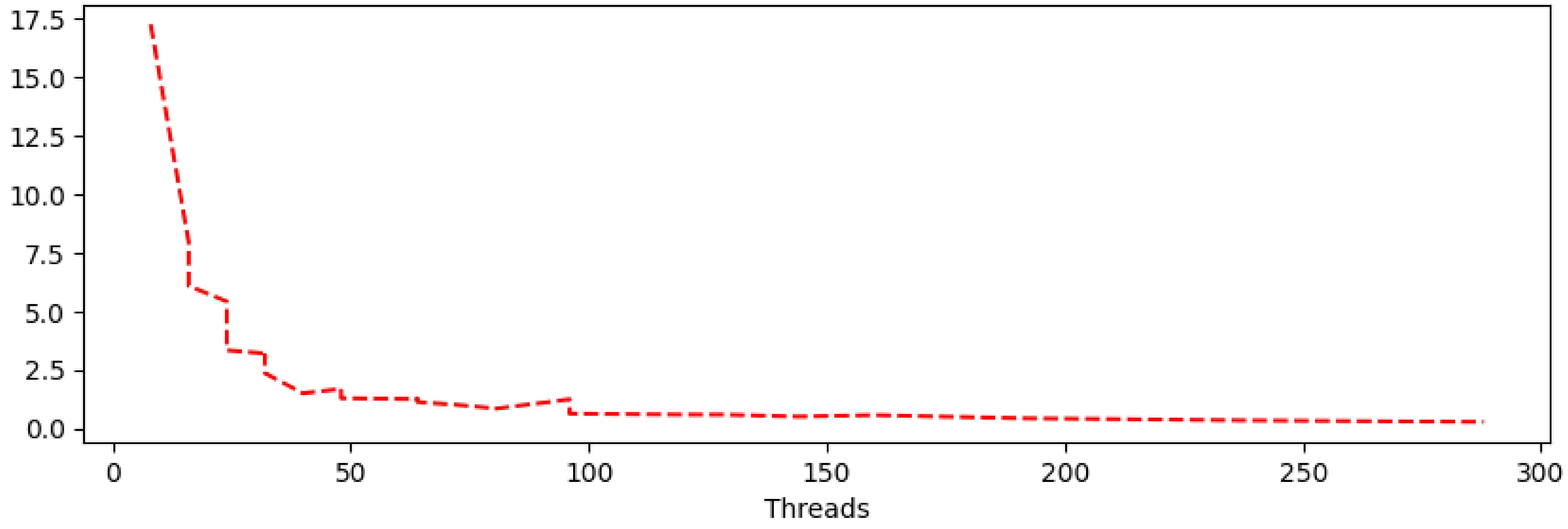
TASK

Use the Nest data frame `df` to:

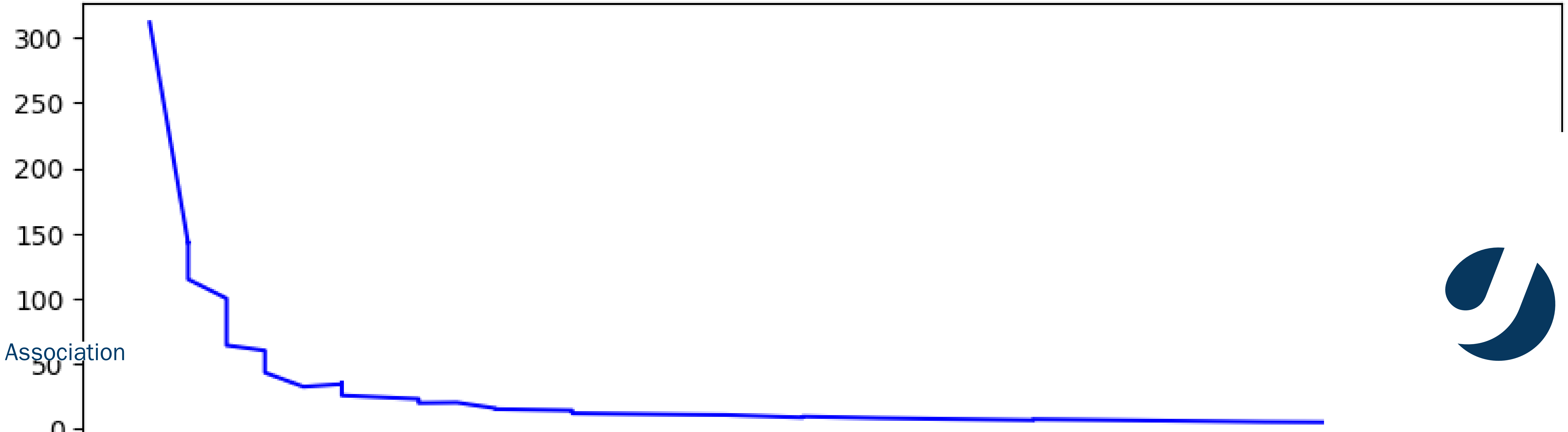
1. Make threads index of the data frame (`.set_index()`)
2. Plot `"Presim. Time / s"` and `"Sim. Time / s"` individually
3. Plot them onto one common canvas!
4. Make them have the same line colors and styles as before
5. Add a legend, add missing axes labels
6. Tell me when you're done with status icon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

```
In [78]: df.set_index("Threads", inplace=True)
```

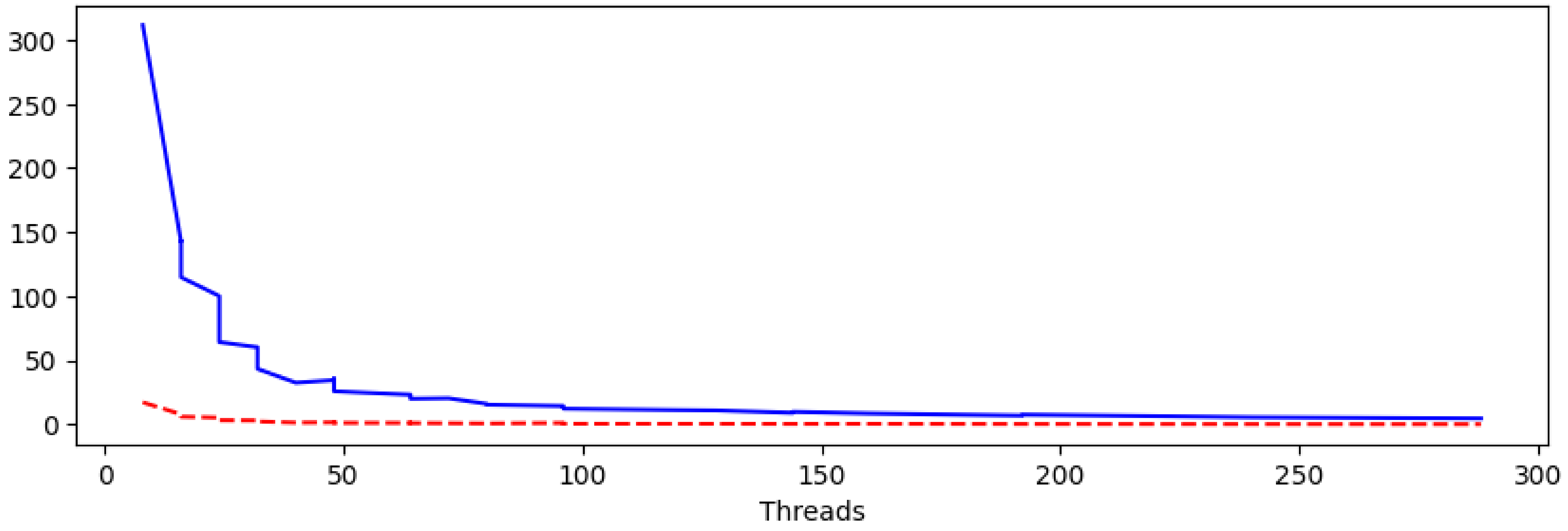
```
In [79]: df["Presim. Time / s"].plot(figsize=(10, 3), style="--", color="red");
```



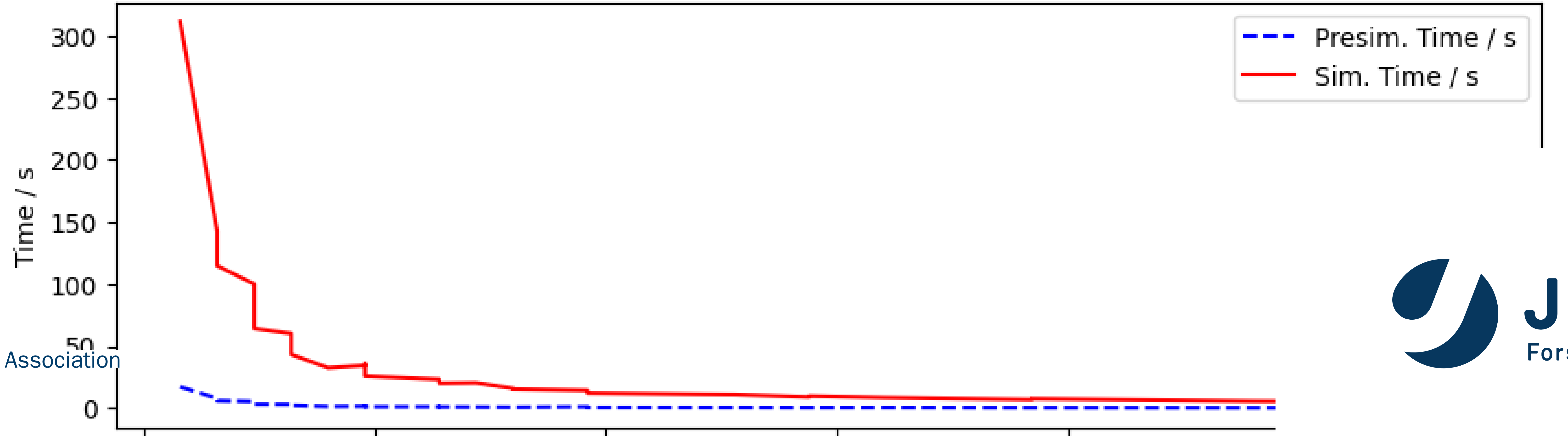

```
In [80]: df["Sim. Time / s"].plot(figsize=(10, 3), style="-b");
```



In [81]:
```
df["Presim. Time / s"].plot(style="--r", figsize=(10,3));
df["Sim. Time / s"].plot(style="-b", figsize=(10,3));
```



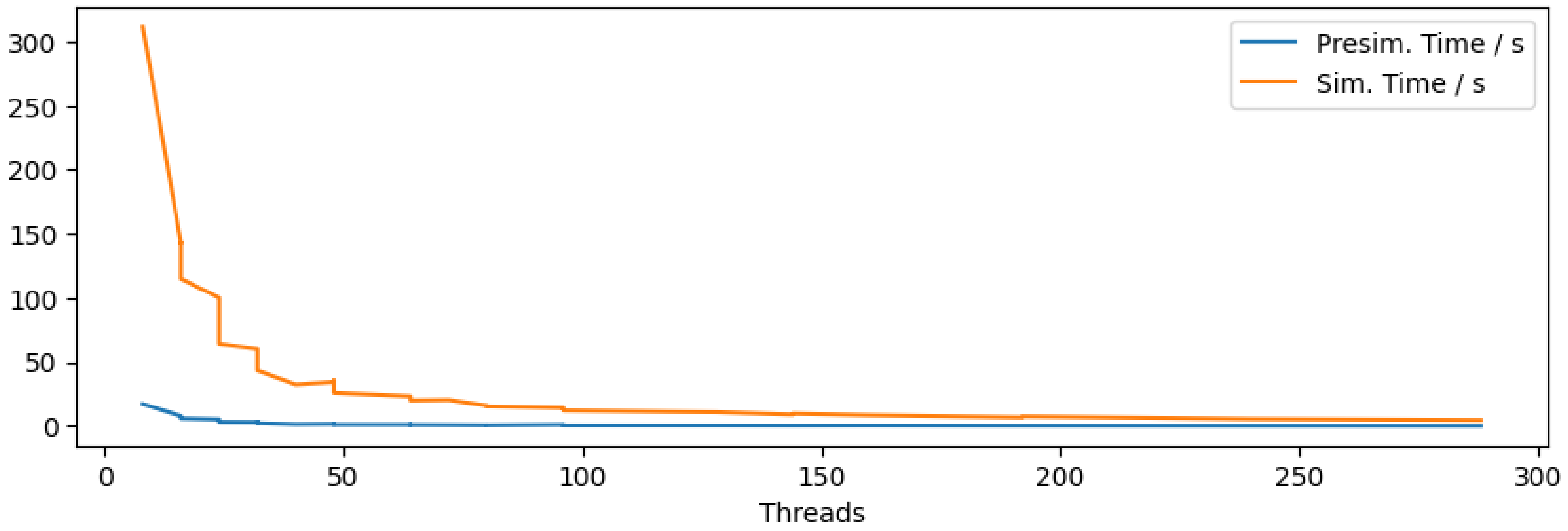
In [82]:
```
ax = df[["Presim. Time / s", "Sim. Time / s"]].plot(style=["--b", "-r"], figsize=(10,3));
ax.set_ylabel("Time / s");
```



MORE PLOTTING WITH PANDAS

Recap: Our first proper Pandas plot

```
In [83]: df[["Presim. Time / s", "Sim. Time / s"]].plot(figsize=(10,3));
```

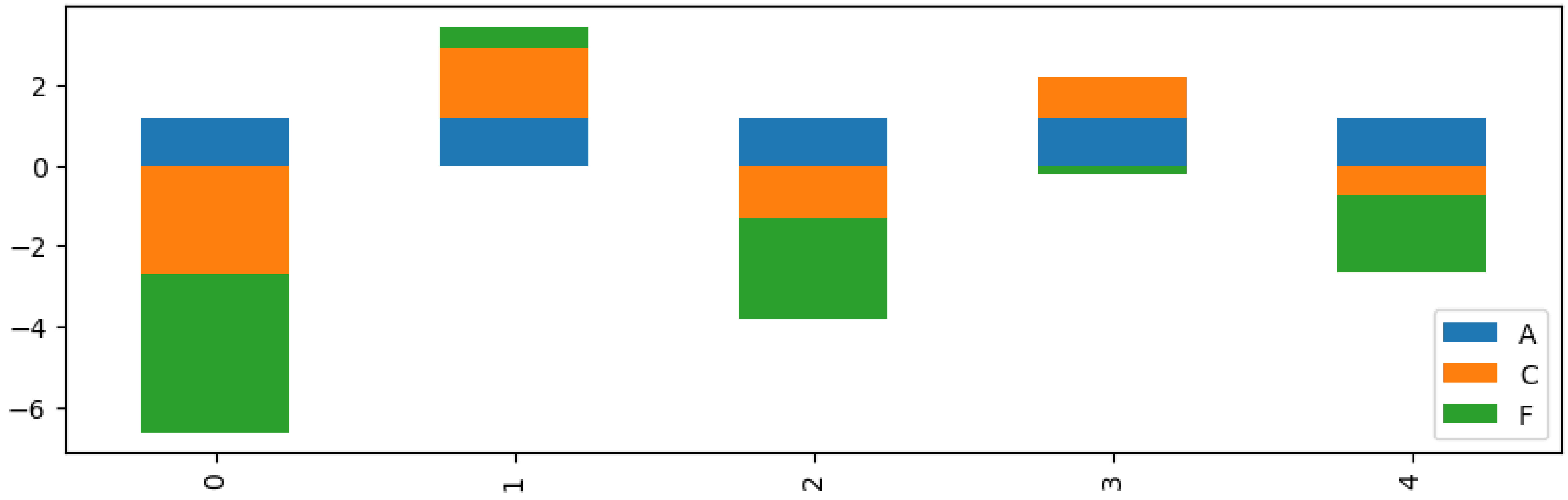


- That's why I think Pandas is great!
- It has great defaults to quickly plot data; basically publication-grade already
- Plotting functionality is very versatile
- Before plotting, data can be *massaged* within data frames, if needed

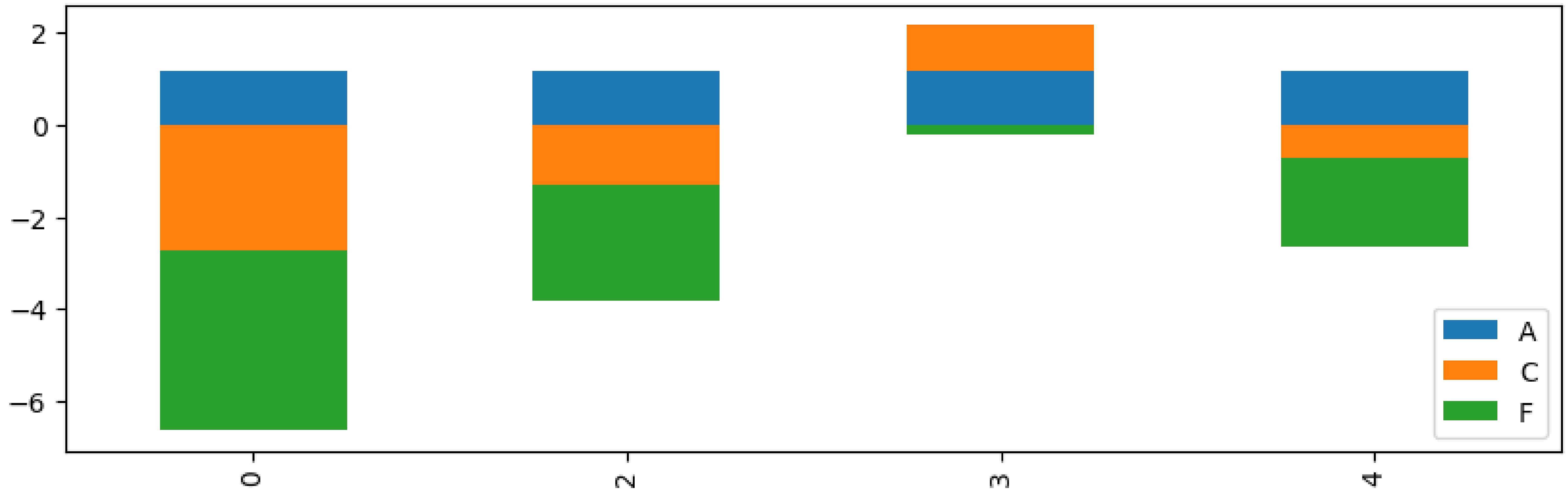
MORE PLOTTING WITH PANDAS

Some versatility

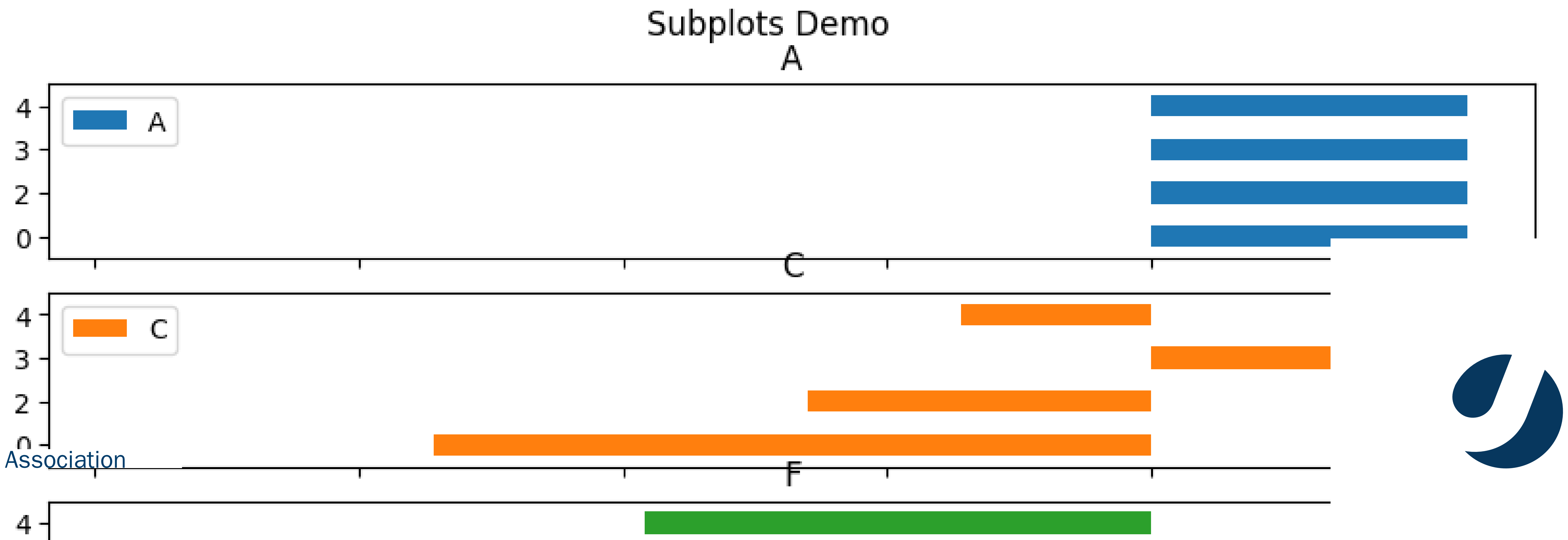
```
In [84]: df_demo[["A", "C", "F"]].plot(kind="bar", stacked=True, figsize=(10,3));
```



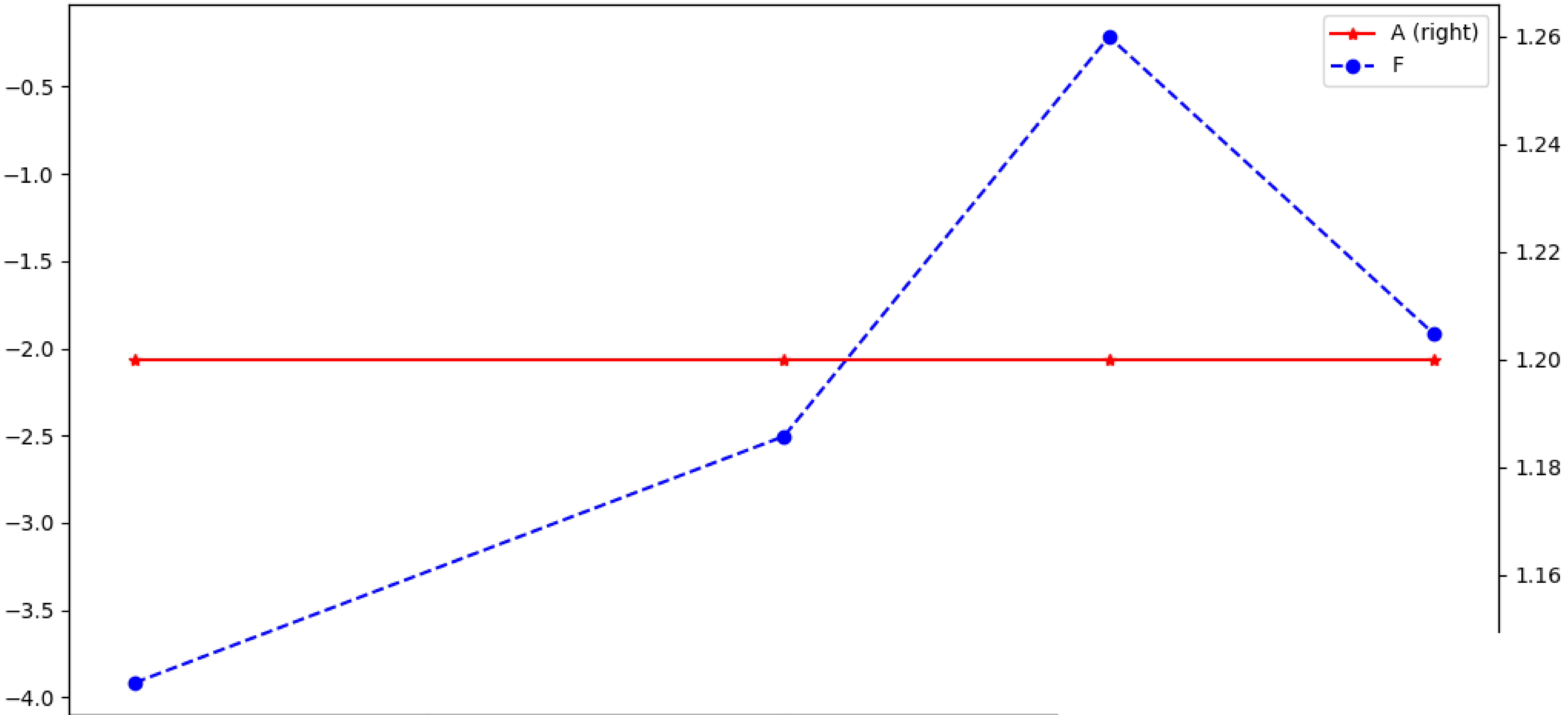
In [85]:
```
df_demo[df_demo["F"] < 0][["A", "C", "F"]].plot(kind="bar", stacked=True, figsize=(10,3));
```



In [86]:
```
df_demo[df_demo["F"] < 0][["A", "C", "F"]]\
    .plot(kind="barh", subplots=True, sharex=True, title="Subplots Demo", figsize=(10, 4));
```

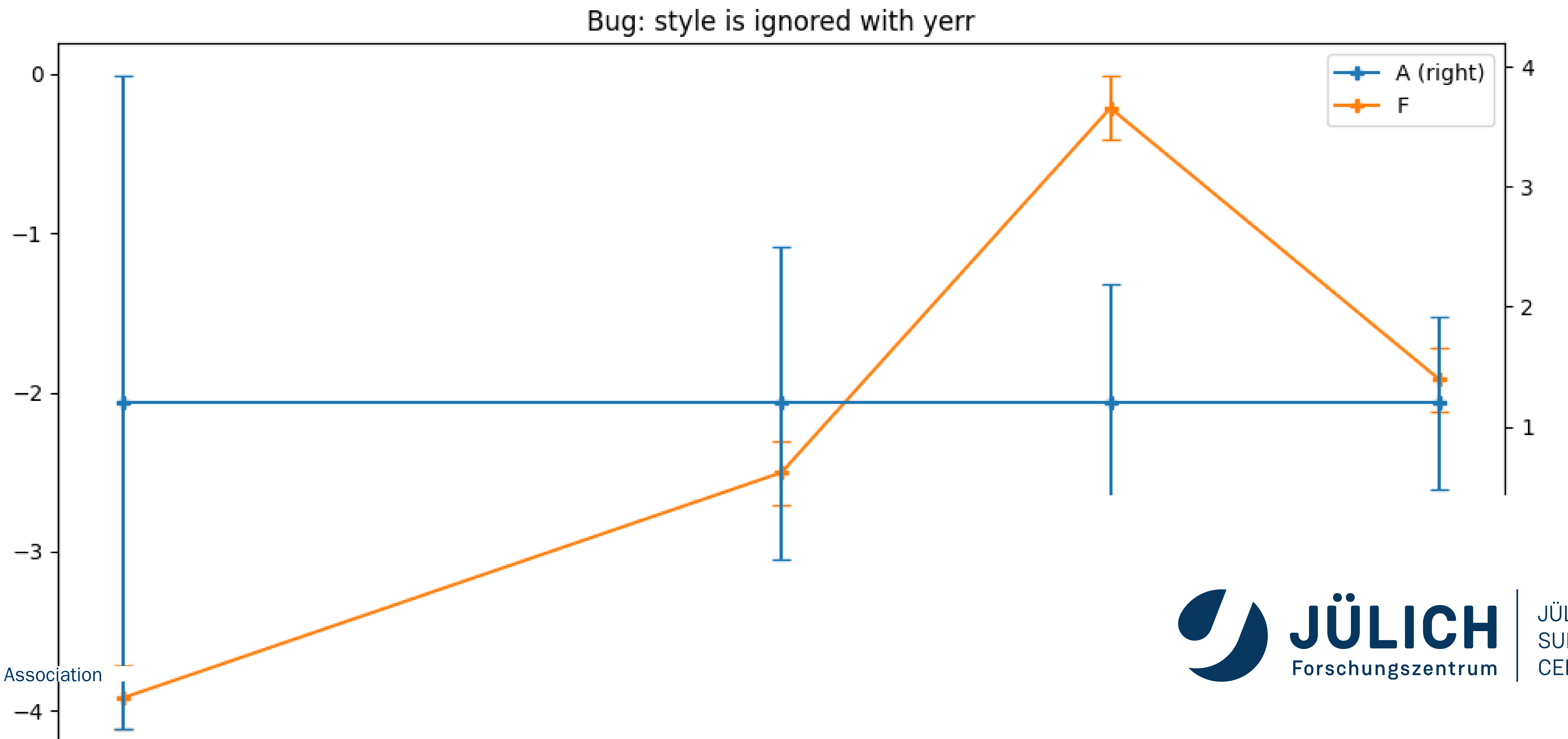


In [87]:
```
df_demo.loc[df_demo["F"] < 0, ["A", "F"]]\
    .plot(
        style=["-*r", "--ob"],
        secondary_y="A",
        figsize=(12, 6),
        table=True
    );
```



	0	2	3
A	1.2	1.2	
F	-3.918281828459045	-2.50406826608595	-0.21376897910983

In [88]:
```
df_demo.loc[df_demo["F"] < 0, ["A", "F"]]\
    .plot(
        style=["-*r", "--ob"],
        secondary_y="A",
        figsize=(12, 6),
        yerr={
            "A": abs(df_demo[df_demo["F"] < 0]["C"]),
            "F": 0.2
        },
        capsize=4,
        title="Bug: style is ignored with yerr",
        marker="P"
    );
```

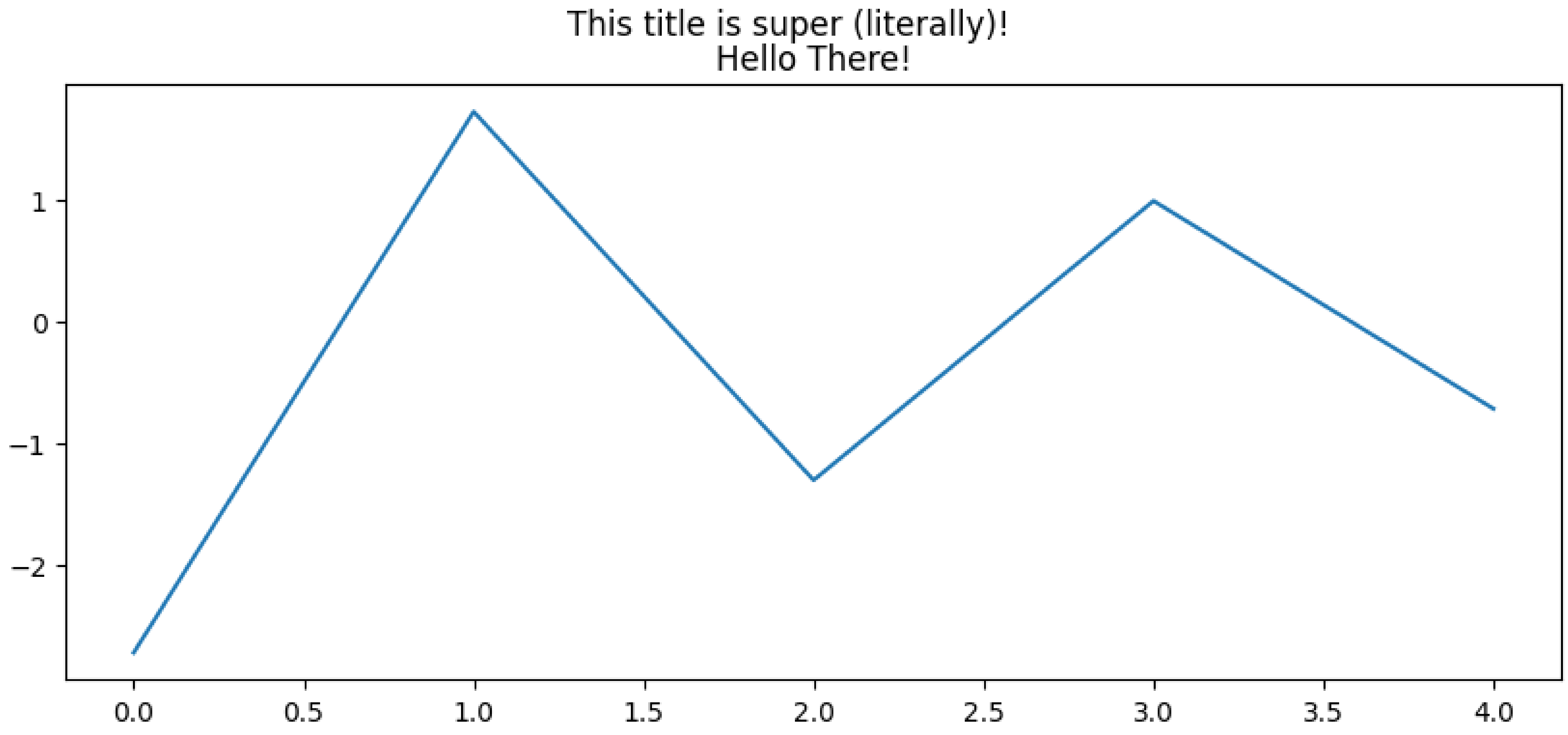


COMBINE PANDAS WITH MATPLOTLIB

- 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

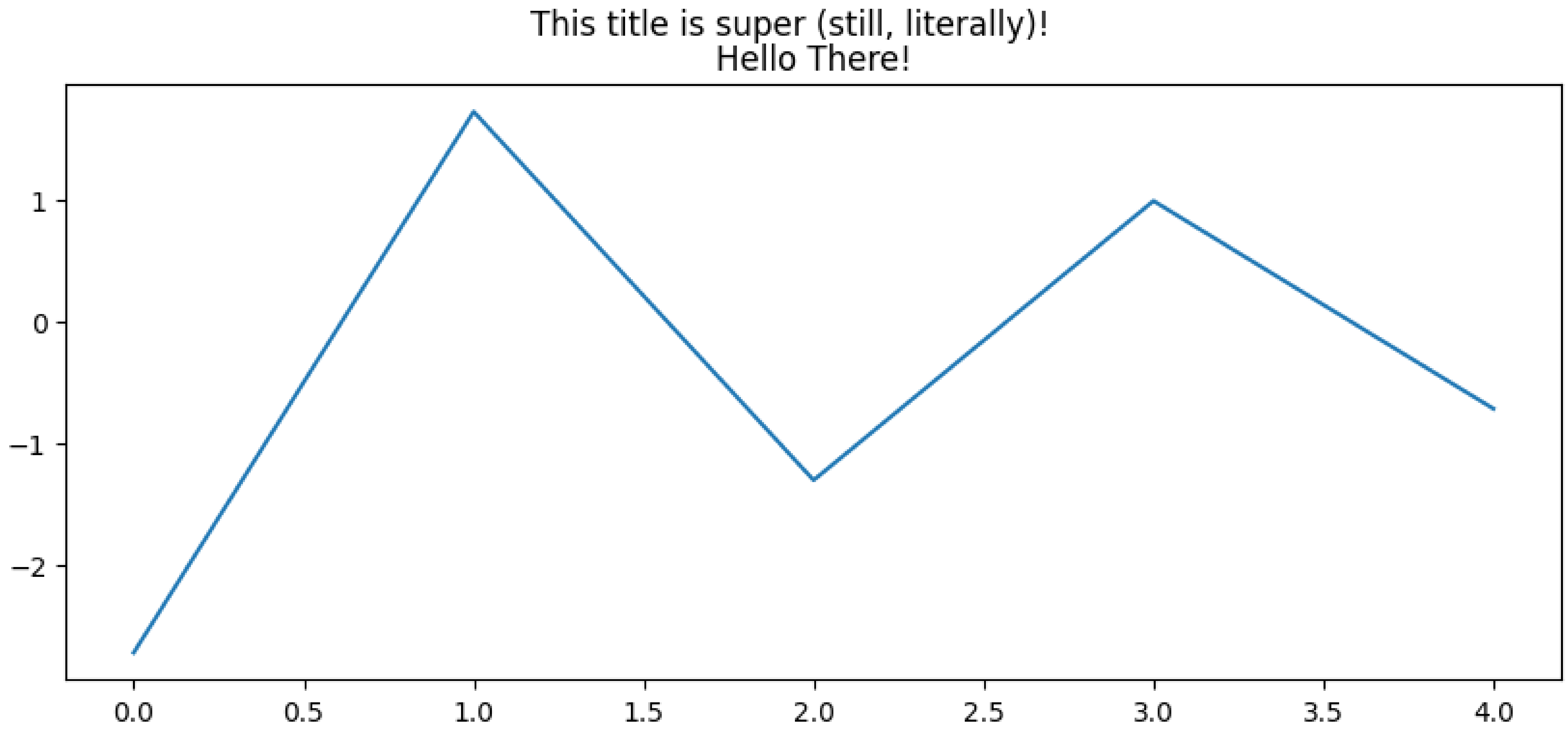
```
In [89]: ax = df_demo["C"].plot(figsize=(10, 4))
         ax.set_title("Hello There!");
         fig = ax.get_figure()
         fig.suptitle("This title is super (literally)!");
```



OPTION 2: DRAW ON MATPLOTLIB AXES

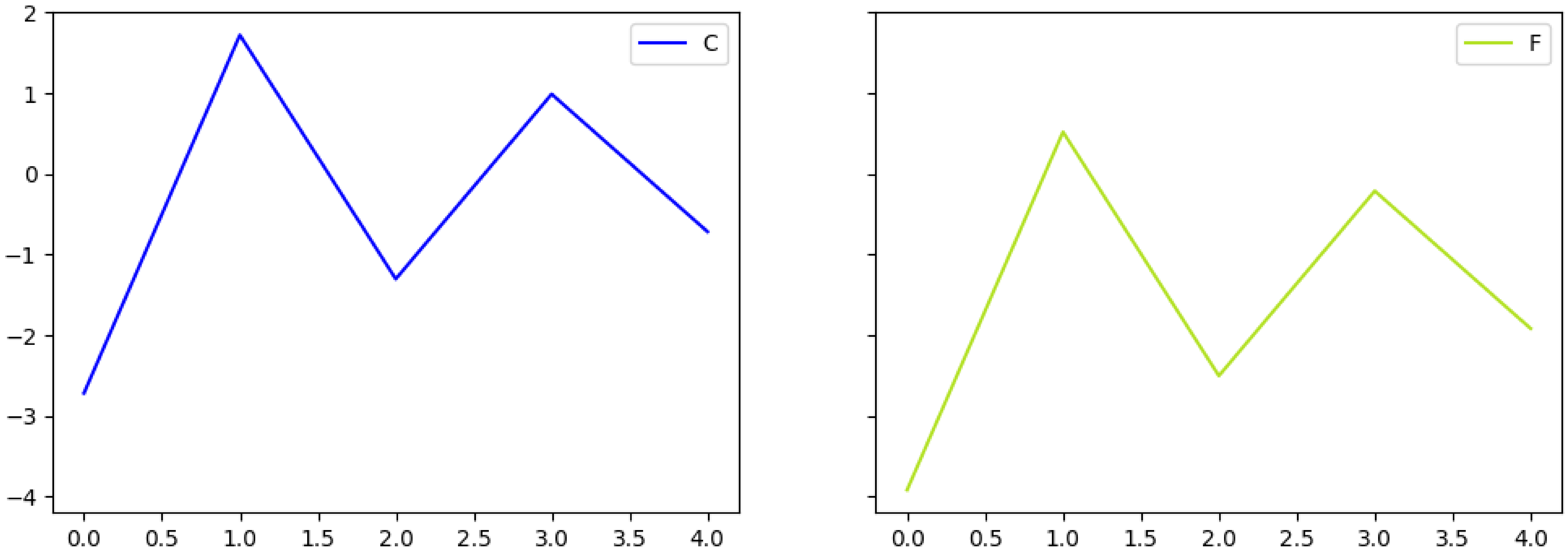
In [90]:

```
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 [91]:
```
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)
```

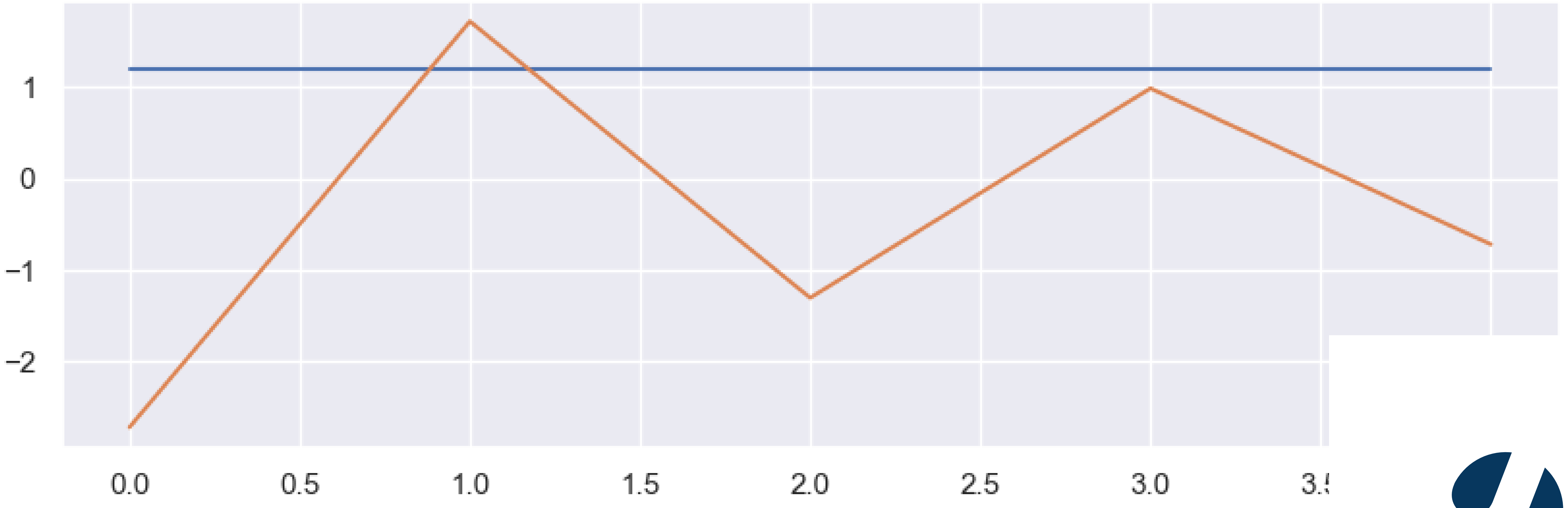


ASIDE: SEABORN

- 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 [92]:
```
import seaborn as sns
sns.set()   # set defaults
```

In [93]:
```
df_demo[["A", "C"]].plot(figsize=(10,3));
```

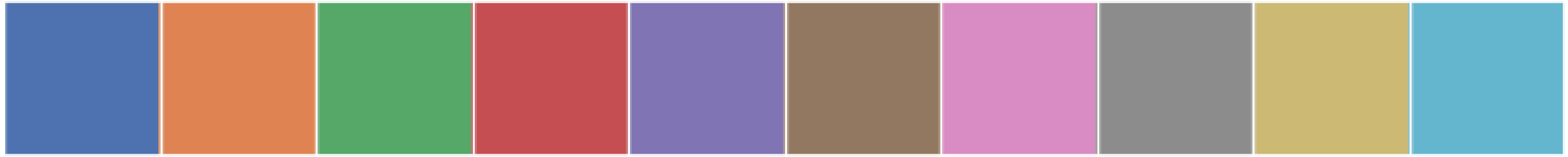


JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

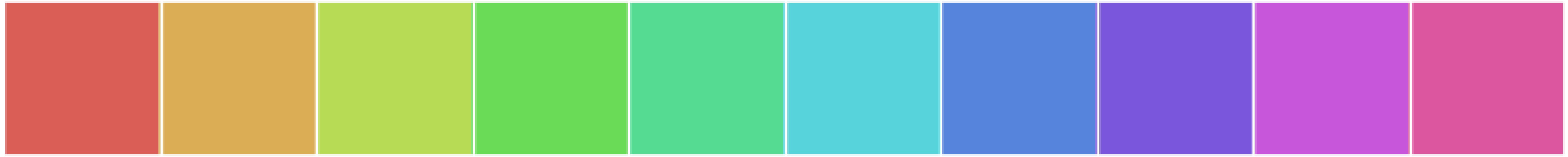
SEABORN COLOR PALETTE EXAMPLE

- Documentation

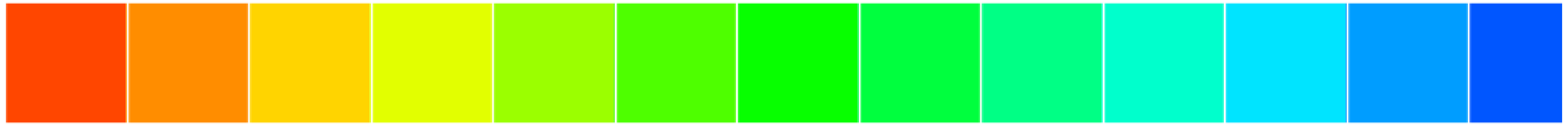
In [94]:
```
sns.palplot(sns.color_palette())
```



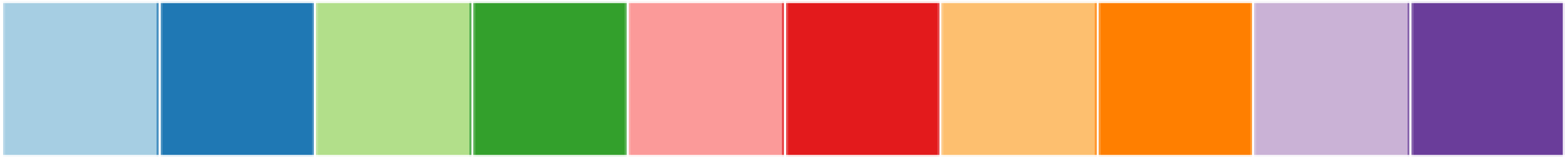
In [95]:
```
sns.palplot(sns.color_palette("hls", 10))
```



In [96]:
```
sns.palplot(sns.color_palette("hsv", 20))
```

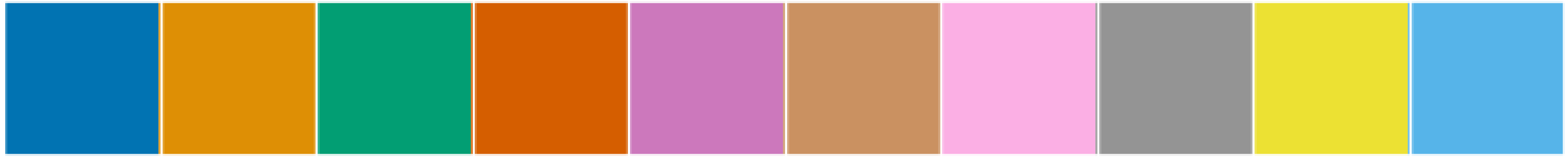


In [97]:
```python
sns.palplot(sns.color_palette("Paired", 10))
```



In [98]:
```python
sns.palplot(sns.color_palette("cubehelix", 8))
```

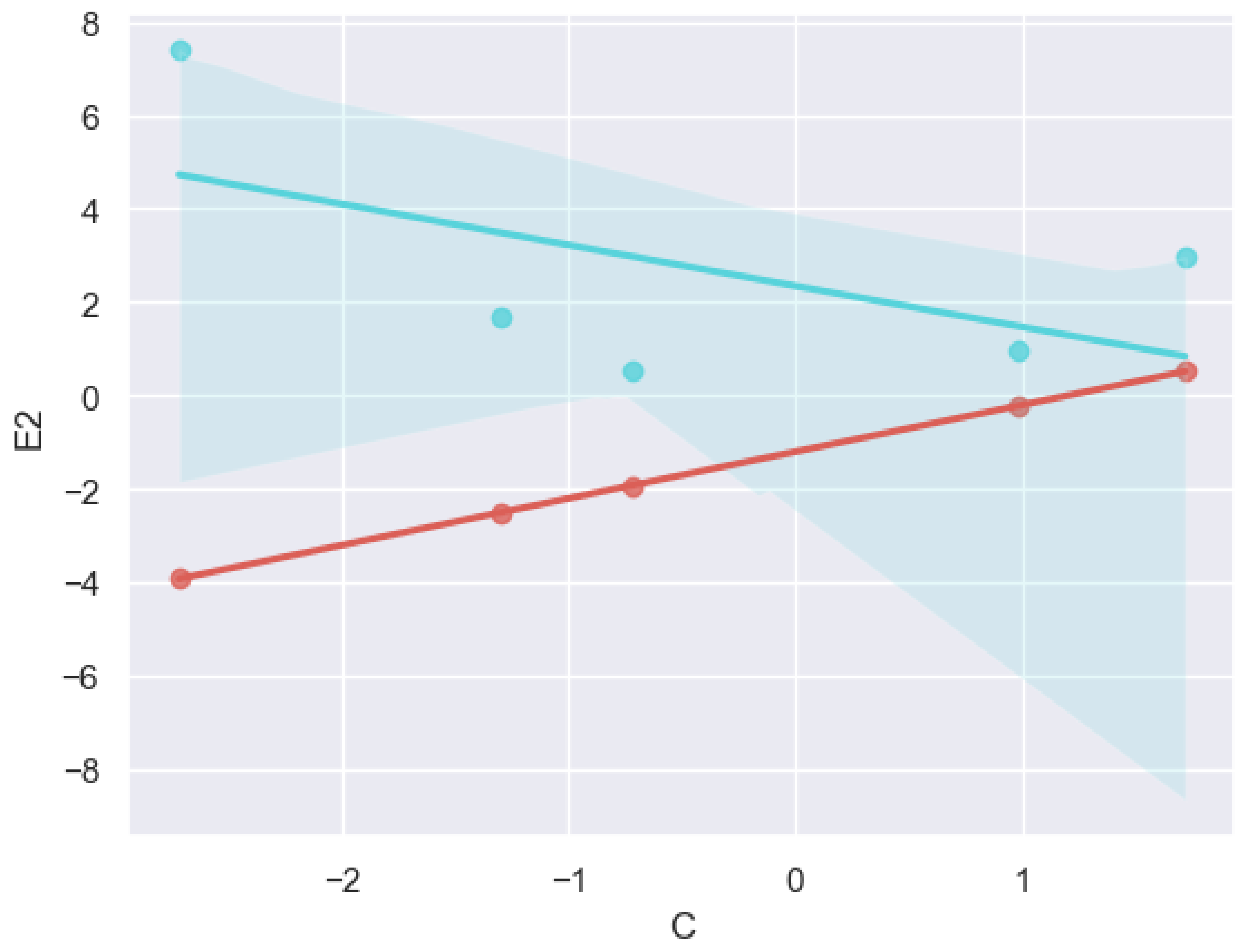
In [99]:
```python
sns.palplot(sns.color_palette("colorblind", 10))
```



SEABORN PLOT EXAMPLES

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

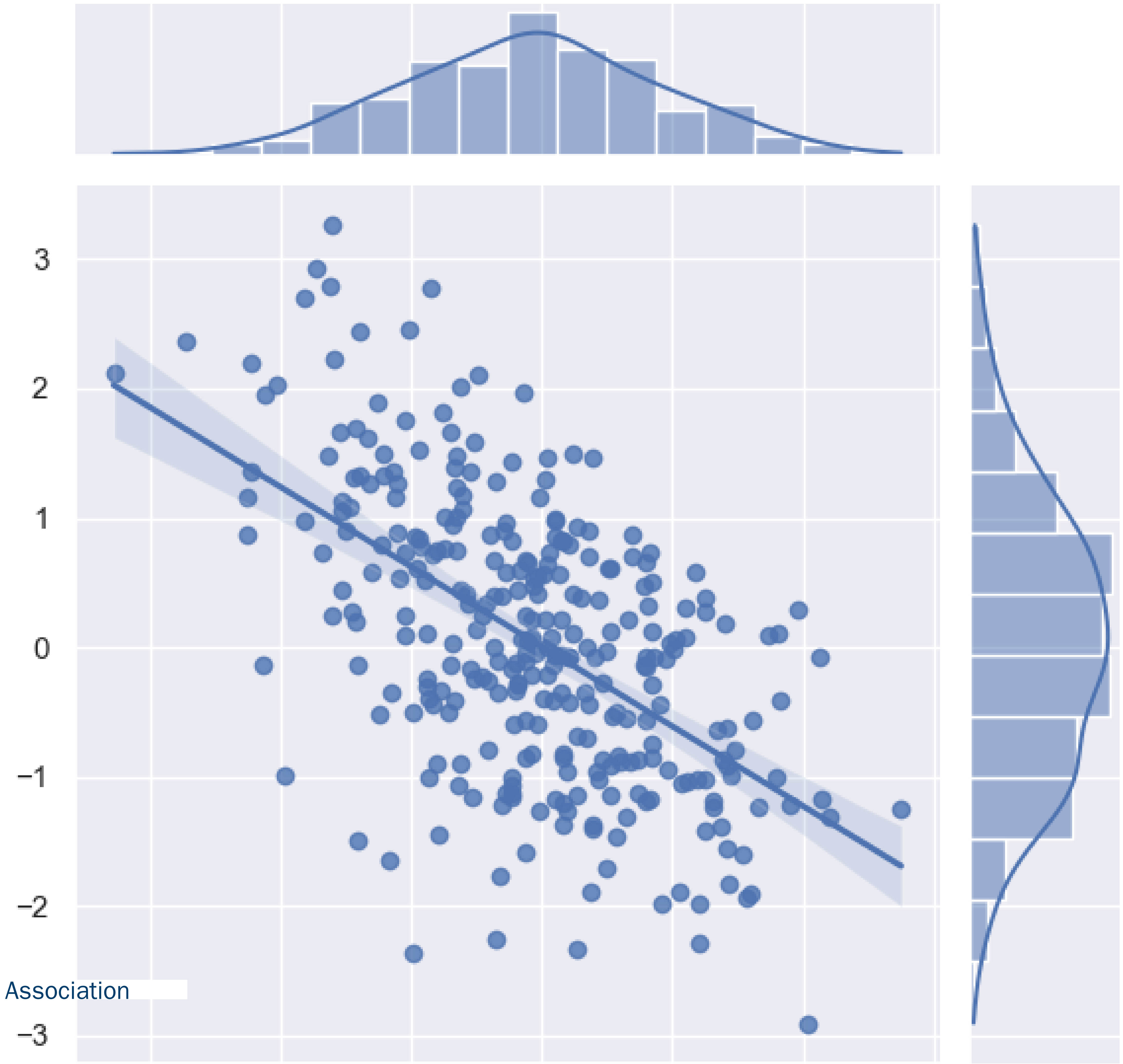
In [100]:
```
with sns.color_palette("hls", 2):
    sns.regplot(x="C", y="F", data=df_demo);
    sns.regplot(x="C", y="E2", data=df_demo);
```



- 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 [101]:
```
x, y = np.random.multivariate_normal([0, 0], [[1, -.5], [-.5, 1]], size=300).T
```

In [102]:
```
sns.jointplot(x=x, y=y, kind="reg");
```



TASK 6

TASK

- 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 [103]:

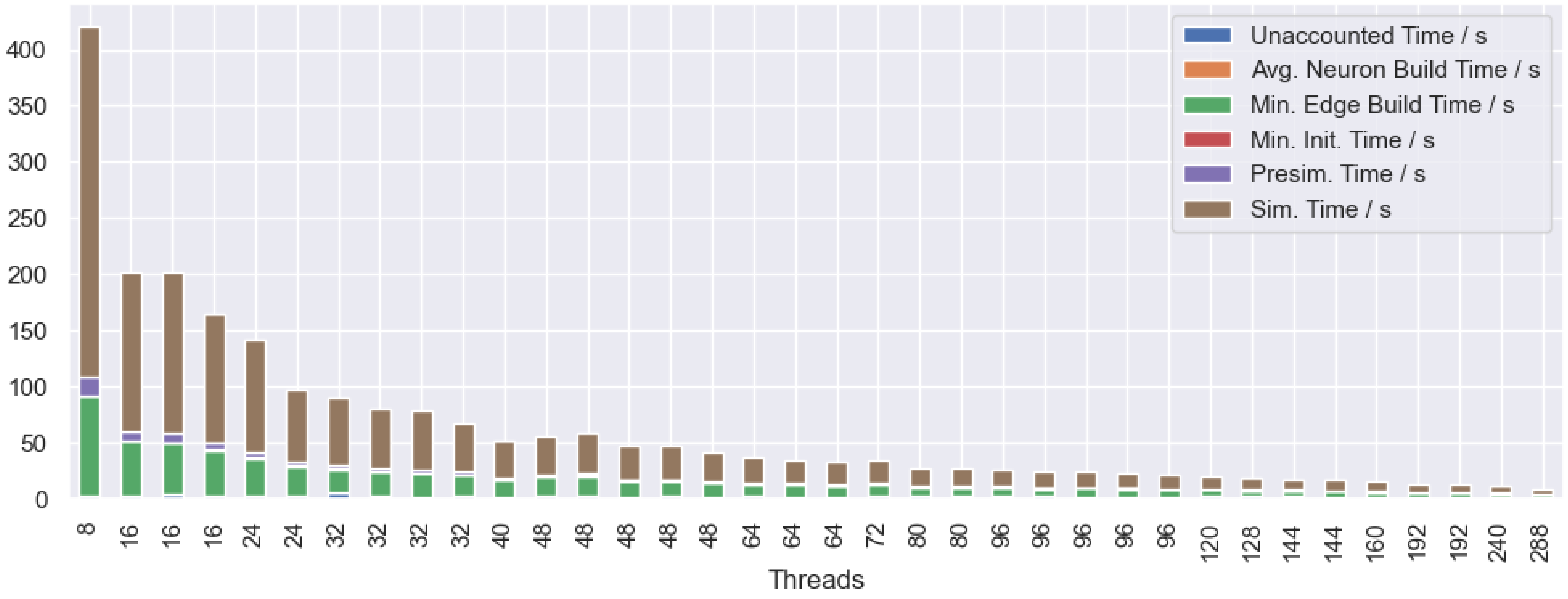
```
cols = [
    'Avg. Neuron Build Time / s',
    'Min. Edge Build Time / s',
    'Min. Init. Time / s',
    'Presim. Time / s',
    'Sim. Time / s'
]
df["Unaccounted Time / s"] = df['Runtime Program / s']
for entry in cols:
    df["Unaccounted Time / s"] = df["Unaccounted Time / s"] - df[entry]
```

In [104]:
```
df[["Runtime Program / s", "Unaccounted Time / s", *cols]].head(2)
```

Out[104]:

Threads	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
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

In [105]:
```
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 anythig, but we use a multi index!

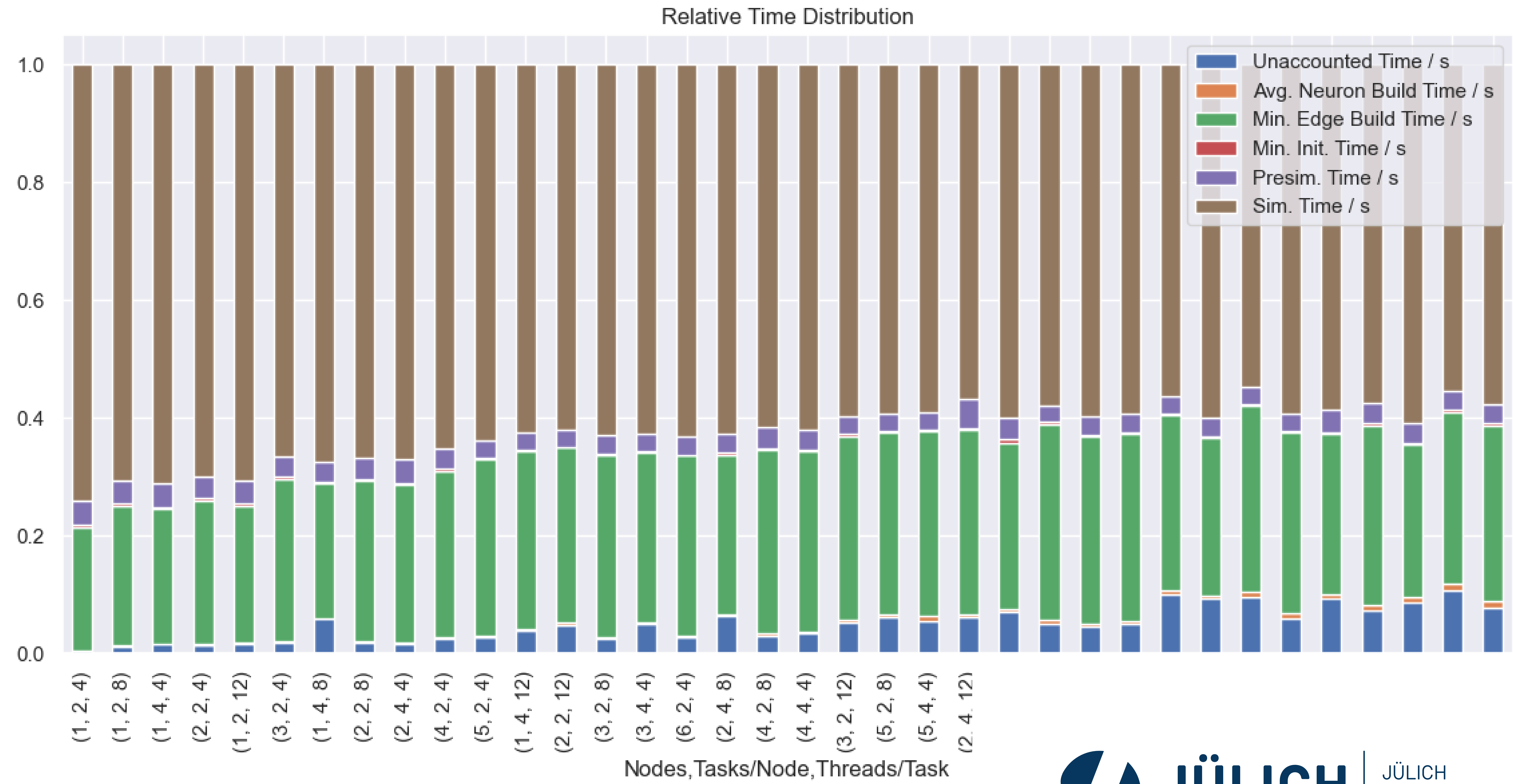
In [106]:
```python
df_multind = df.set_index(["Nodes", "Tasks/Node", "Threads/Task"])
df_multind.head()
```

Out[106]:

Nodes	Tasks/Node	Threads/Task	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	Num Conne
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	126573
		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	126573
	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	126573
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	126573
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	126573

In [107]:

```
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)

GROUPING

- Group a frame by common values of column(s)
- Use operations on this group
- Grouped frame is not *directly* a new frame, but only through an applied operation

In [108]:
```
df.groupby("Nodes").groups
```

Out[108]:
```
{1: [8, 16, 16, 24, 32, 48], 2: [16, 32, 32, 48, 64, 96], 3: [24, 48, 48, 72, 96, 144], 4: [32, 64, 64, 96, 12
8, 192], 5: [40, 80, 80, 120, 160, 240], 6: [48, 96, 96, 144, 192, 288]}
```

In [109]:
```
df.groupby("Nodes").get_group(4).head(3)
```

Out[109]:

	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	Numb Connec
Threads																		
32	5	4	2	4	66.58	10	True	0.13	18.86	19.65	...	2.35	43.38	47361344.0	821491	7.23	112500	126573
64	5	4	2	8	34.09	10	True	0.14	10.60	10.83	...	1.25	20.96	47074752.0	818198	7.33	112500	126573
64	5	4	4	4	32.49	10	True	0.09	9.98	10.31	...	1.12	20.12	48081056.0	818198	7.33	112500	126573

3 rows × 22 columns

In [110]:
```
df.groupby("Nodes").mean()
```

Out[110]:

	id	Tasks/Node	Threads/Task	Runtime Program / s	Scale	Plastic	Avg. Neuron Build Time / s	Min. Edge Build Time / s
Nodes								

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

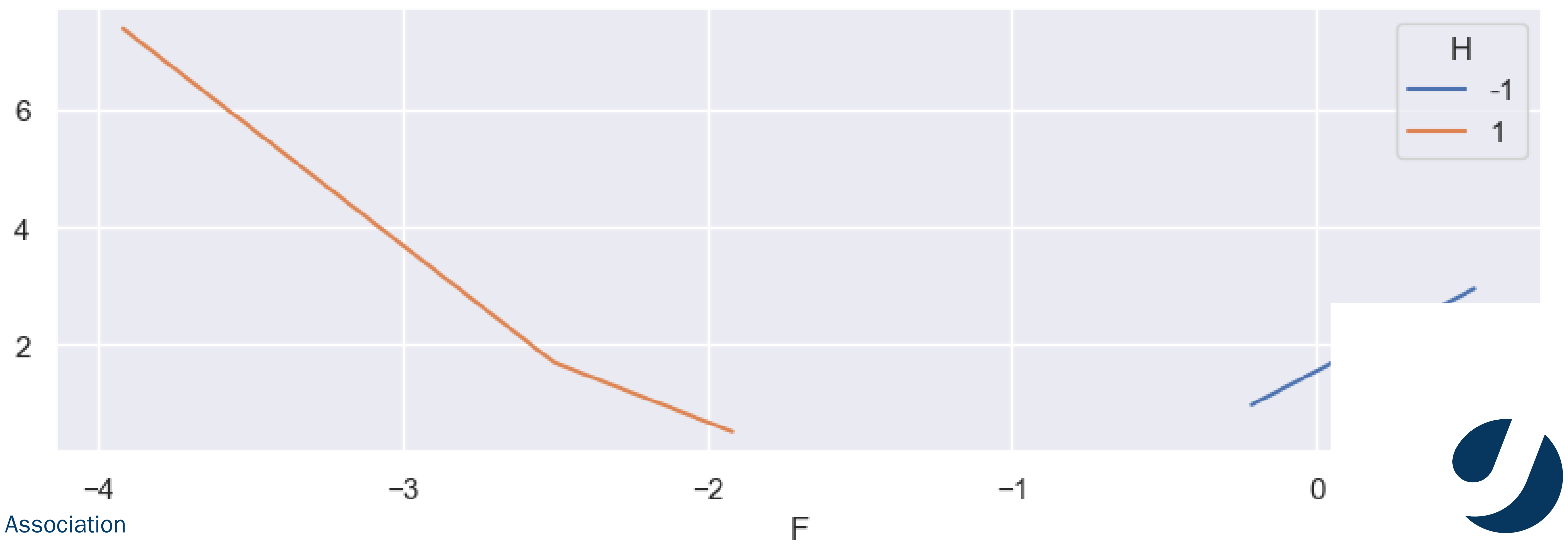
In [111]:
```
df_demo["H"] = [(-1)**n for n in range(5)]
```

In [112]:
```
df_pivot = df_demo.pivot_table(
    index="F",
    values="E2",
    columns="H"
)
df_pivot
```

Out[112]:

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 [113]:
```
df_pivot.plot(figsize=(10,3));
```

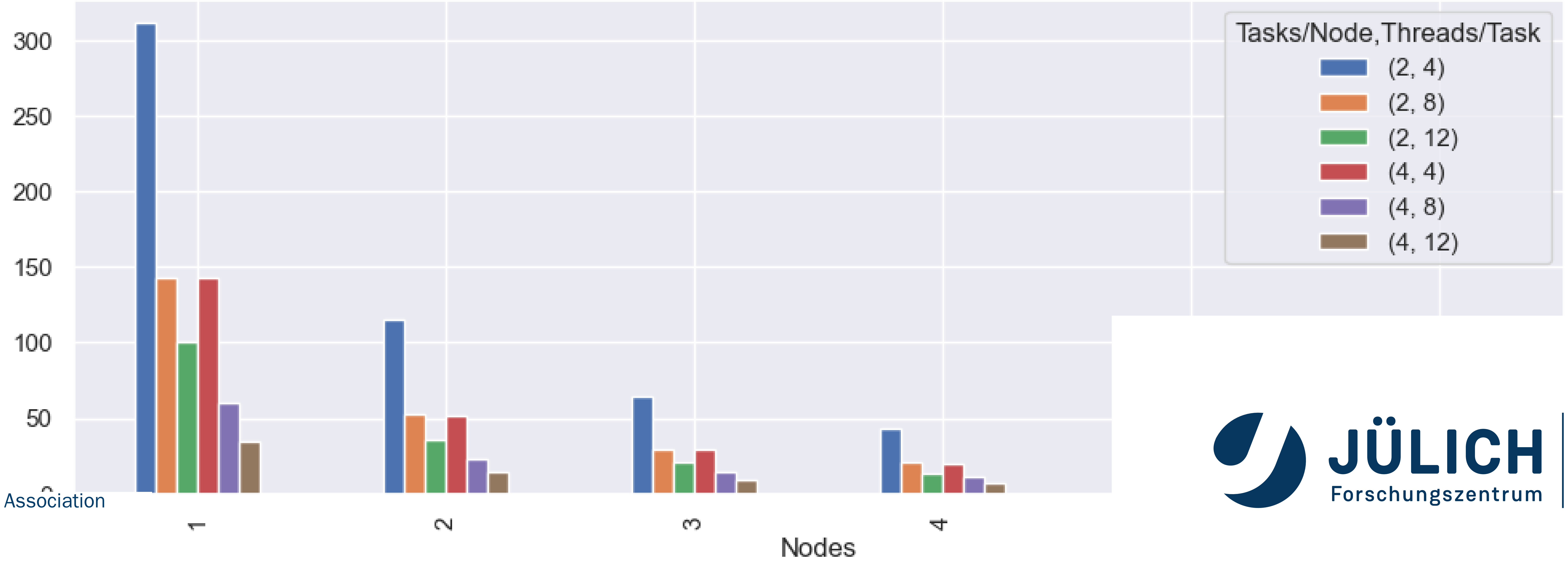


TASK 7

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 [114]:
```
df.pivot_table(
    index="Nodes",
    columns=["Tasks/Node", "Threads/Task"],
    values="Sim. Time / s",
).plot(kind="bar", figsize=(12, 4));
```



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

TASK 7B (LIKE *B*ONUS)

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!

PANDAS 2

- Pandas 2.0 was released in April 2023
- Only limited deprecations (i.e. *an upgrade is probably safe*)
- Key new feature: Apache Arrow support (via PyArrow)
- Fine-grained installation options `python3 -m pip install 'pandas[performance, excel]'`

* Get a reasonably large data source (larger would be better, though)
* Example: Train stations as provided by Deutsche Bahn

In [115]:
```
data_db = 'db-bahnhoefe.csv'  # source: https://download-data.deutschebahn.com/static/datasets/stationsdaten/DBSuS-Uebersicht_Bahnhoefe-Stand2
```

In [116]:
```
%timeit pd.read_csv(data_db, sep=';')
```

```
8.97 ms ± 320 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [117]:
```
%timeit pd.read_csv(data_db, sep=';', engine='pyarrow', dtype_backend='pyarrow')
```

```
2.54 ms ± 84.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

LARGE DATA & MANGLING

- Pandas can read data directly in `tar` form
- Pandas can read data directly from online resource
- Let's combine that to an advanced task!
- It works also with the PyArrow backend (remember to download the online resource when testing; there is no cache!)

TASK 8 (SUPER BONUS)

TASK

- Create bar chart of top 10 actors (on `x`) and average ratings of their top movies (`y`) based on IMDb data (only if they play in at least two movies)

- IMDb provides data sets at datasets.imdbws.com
- Can directly be loaded like

```
pd.read_table('https://datasets.imdbws.com/dataset.tsv.gz', sep="\t", low_memory=False, na_values=["\\N","nan"])
```

- Needed:
  - `name.basics.tsv.gz` (for names of actors and movies they are known for)
  - `title.ratings.tsv.gz` (for ratings of titles)
- Strategy *suggestions*:
  - Use `df.apply()` with custom function
  - Custom function: Compute average rating and determine if this entry is eligible for plotting (this *can* be done at once, but does not need to be)
  - Average rating: Look up title IDs as listed in `knownForTitles` in titles dataframe

```
df_names = pd.read_table('imdb-data/name.basics.tsv.gz', sep="\t", low_memory=False, na_values=["\\N","nan"])
df_ratings = pd.read_table('https://datasets.imdbws.com/title.ratings.tsv.gz', sep="\t", low_memory=False, na_values=["\\N","nan"])

df_names_i = df_names.set_index('nconst')
df_ratings_i = df_ratings.set_index('tconst')

df_names_i = pd.concat(
    [
        df_names_i,
        df_names_i.apply(lambda line: valid_and_avg_rating(line), axis=1, result_type='expand')
    ]
    , axis=1
)
df_names_i[df_names_i['toPlot'] == True].sort_values('avgRating', ascending=False).iloc[0:10].reset_index().set_index('primaryName')
['avgRating'].plot(kind='bar')
```

```python
def valid_and_avg_rating(row):
    rating = 0
    ntitles = 0
    _titles = row['knownForTitles']
    _professions = row['primaryProfession']
    if not isinstance(_titles, str):
        _titles = str(_titles)
    if not isinstance(_professions, str):
        _professions = str(_professions)
    titles = _titles.split(',')
    professions = _professions.split(',')
    for title in titles:
        if title in df_ratings_i.index:
            rating += df_ratings_i.loc[title]['averageRating']
            ntitles += 1
    if ntitles > 0:
        plot = False
        if ntitles > 2:
            if 'actor' in professions:
                plot = True
        return {'toPlot': plot, 'avgRating': rating / ntitles}
    else:
        return {'toPlot': False, 'avgRating': pd.NA}
```

TASK 8B (*B*ONUSEPTION)

TASK

All of the following are ideas for unique sub-tasks, which can be done individually

- In addition to Task 8, restrict the top titles to those with more than 10000 votes
- For 30 top-rated actors, plot rating vs. age
- For 30 top-rated actors, plot rating vs. average runtime of the known-for-titles (using `title.basics.tsv.gz`)

RANDOM FEATURES NOT SHOWN

This are all links:

- `df.drop()`
- `df.corr()`
- `df.boxplot()`
- `pd.read_sql_query("SELECT * FROM purchases", con)`
- `df.duplicated()` and `df.drop_duplicates()`
- Aliases for categorical data
- Working with time
    - `ts.tz_convert`
    - `pd.period_range()`
    - `pd.period_range().asfreq()`

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
    - Whats new in Pandas 2