

09- Data analysis and drawing

Now let's bite into the bigger bite: weather data.

We'll use a data file in this format:

```
print(open("vremenske-postaje.txt").read()[:300])
```

```
Murska Sobota,1961-01-01,-1,-4
Murska Sobota,1961-01-02,1,-1
Murska Sobota,1961-01-03,7,-4
Murska Sobota,1961-01-04,7,3
Murska Sobota,1961-01-05,4,-0
Murska Sobota,1961-01-06,2,-2
Murska Sobota,1961-01-07,5,0
Murska Sobota,1961-01-08,5,-1
Murska Sobota,1961-01-09,4,-1
Murska Sobota,1961-01-10,4,-1
M
```

Read with `np.genfromtxt`; tell it to delimit the data with commas (`delimiter=","`) and to read it as strings of up to 20 characters (`dtype="U20"`).

```
import numpy as np
```

```
podatki = np.genfromtxt(open("/Users/janez/Downloads/vremenske-postaje.txt"), delimiter=",", dtype="U20")
```

```
podatki[:5]
```

```
array(['Murska Sobota', '1961-01-01', '-1', '-4'],
      ['Murska Sobota', '1961-01-02', '1', '-1'],
      ['Murska Sobota', '1961-01-03', '7', '-4'],
      ['Murska Sobota', '1961-01-04', '7', '3'],
      ['Murska Sobota', '1961-01-05', '4', '-0']], dtype='<U20')
```

As in the stock exchange minutes, we turn the columns into rows and store them in separate variables.

```
↳ ↵ ↻
```

```
kraji, datumi, tmax, tmin = podatki.T
```

If we are only interested in data for Ljubljana, let's build a mask.

```
maska = kraji == "Ljubljana"
```

For simplicity, let's now just throw out the data for the other places.

```
datumi = datumi[maska]
tmax = tmax[maska]
tmin = tmin[maska]
```

Ah, that's so annoying. It would be simpler to apply a mask to the whole table and then unpack. Repeat the exercise.

```
podatki = np.genfromtxt(open("/Users/janez/Downloads/vremenske-postaje.txt"), delimiter=";", dtype="U20")
maska = podatki[:, 0] == "Ljubljana"
_, datumi, tmax, tmin = podatki[maska].T
tmax = tmax.astype(float)
tmin = tmin.astype(float)
```

We don't care about the location (it's always Ljubljana anyway), so we just saved it in `_`.

Now comes the fun part: we'd like a table with the (maximum) temperatures for each day. The table would have three indices: year, month and day. So `temp[71, 0, 25]` would be the temperature on 26 January 71. Obviously, this will be a three-dimensional table. Let's prepare it: at the beginning it should contain only `np.nan`, not a number.

```
temp = np.full((124, 12, 31), np.nan)
```

Now it needs to be filled with data. The date column will need to be unpacked.

```
datumi
```

```
[8]:
```

```
array(['1900-01-01', '1900-01-02', '1900-01-03', ..., '2023-10-10',
       '2023-10-11', '2023-10-13'], dtype='<U20')
```

At first glance: you want the whole table (all strings), but only the first four characters for each.

```
datumi[:, :4]
```

```
-----
IndexError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 datumi[:, :4]

IndexError: too many indices for array: array is 1-dimensional, but 2 were indexed
```

No, this table is one-dimensional and does not allow a second index. Numpy functions for working with array tables are in `np.char`. The view stops at `np.char.split`, but this doesn't make us happy, because it returns a table of lists, and we can't help ourselves with them.

```
np.char.split(datumi, "-")
```

```
[10]:
```

```
array([list(['1900', '01', '01']), list(['1900', '01', '02']),
       list(['1900', '01', '03']), ..., list(['2023', '10', '10']),
       list(['2023', '10', '11']), list(['2023', '10', '13'])],
      dtype=object)
```

Partition saves us. (There may be something else, simpler. This part of numpy is not my strong area.)

```
np.char.partition(datumi, "-")
```

```
[11]:
```

```
array([[ '1900', '-', '01-01'],  
      [ '1900', '-', '01-02'],  
      [ '1900', '-', '01-03'],  
      ...,  
      [ '2023', '-', '10-10'],  
      [ '2023', '-', '10-11'],  
      [ '2023', '-', '10-13']], dtype='<U5')
```

Python arrays also have a partition method, and the result is the same: three arrays - everything before the character you're partitioning by, then that character, and then everything after it. Here, in numpy, instead of three strings, we get three columns of strings. We invert, we break into rows. We divide the last one again.

```
leto, _, mes_dan = np.char.partition(datumi, "-").T  
mes, _, dan = np.char.partition(mes_dan, "-").T  
  
leto = leto.astype(int)  
mes = mes.astype(int)  
dan = dan.astype(int)
```

This gives us three lists containing all the years, months and dates from all the lines in the file. The corresponding temperatures are in tmax, also by line. And now we can overwrite them in the temp table.

```
temp[leto - 1900, mes - 1, dan - 1] = tmax
```

We subtract 1900 from the year, and 1 from each month and day, because for some reason people don't count years from 1900 onwards, and months and days from 0.

Here we have it: the maximum daily temperature on 26 January 1971 was

```
temp[71, 0, 25]
```

```
[14]:
```

```
2.0
```

What have been the average monthly temperatures since 2019?

We take the temp[119:] (because we subtracted 1900 years) and calculate the nanmean on axis 2. Axis 2 is the days of the month - we want to calculate the average over them.

```
np.nanmean(temp[119:], axis=2)
```

```
/var/folders/2y/4j70c4q56811j41b6g1r0fk00000gn/T/ipykernel_56101/117835579.py:1: RuntimeWarning: Mean of empty slice  
np.nanmean(temp[119:], axis=2)
```

```
array([[ 4.          , 11.32142857, 15.16129032, 16.73333333, 17.70967742,  
        29.4          , 28.93548387, 28.32258065, 22.3          , 18.67741935,  
        11.16666667,  7.06451613],  
       [ 6.74193548, 11.75862069, 12.77419355, 20.1          , 20.74193548,  
        24.86666667, 27.93548387, 28.16129032, 23.43333333, 16.61290323,  
         9.2          ,  4.48387097],  
       [ 4.09677419, 10.67857143, 13.41935484, 14.8          , 19.32258065,  
        29.23333333, 29.06451613, 27.35483871, 24.3          , 14.83870968,  
         8.63333333,  3.4516129 ],  
       [ 5.4516129 , 10.39285714, 13.32258065, 16.13333333, 24.29032258,  
        29.86666667, 30.90322581, 29.22580645, 21.53333333, 20.51612903,  
        11.23333333,  6.29032258],  
       [ 6.74193548,  9.07142857, 13.90322581, 15.9          , 20.5483871 ,  
        26.76666667, 28.83870968, 27.93548387, 26.94444444, 23.63636364,  
         nan          , nan]])
```

The above warning refers to the last two items: we don't have data for November and December 2023 (I picked up a 7 GB file with data for the whole world in October 2023), so nanmean has returned nan there.

What about average monthly temperatures in general?

In this case we average over years and days, only the months need to remain. So the axes will be 0 and 2.

```
monthly = np.nanmean(temp, axis=(0, 2))  
  
monthly_min = np.nanmin(temp, axis=(0, 2))  
monthly_max = np.nanmax(temp, axis=(0, 2))
```

```
[28]:
```

```
monthly
```

```
[28]:
```

```
array([ 2.38911129,  5.54746462, 10.78914687, 15.72681704, 20.73475936,  
        24.4570011 , 26.72576326, 26.07096774, 21.48240223, 15.41874323,  
         8.40751121,  3.43990188])
```

We know better.

```
import locale
import calendar

locale.setlocale(locale.LC_ALL, "sl_SI")

for month, t in zip(calendar.month_name[1:], monthly):
    print(f"{month.title():>10}: {t:4.1f} {'*' * int(t)}")

    Januar:  2.4 **
    Februar:  5.5 *****
    Marec: 10.8 *****
    April: 15.7 *****
    Maj: 20.7 *****
    Junij: 24.5 *****
    Julij: 26.7 *****
    Avgust: 26.1 *****
    September: 21.5 *****
    Oktober: 15.4 *****
    November:  8.4 *****
    December:  3.4 ***
```

OK, enough fiddling: isn't it time we drew a real graph of temperatures?

Drawing graphs

There are, of course, several libraries for drawing graphs, which work in different environments. For our purposes, the simplest one is the one that gets along well with Jupyter Notebook. (Incidentally, I must admit that I don't draw in Notebook myself, but elsewhere with other libraries, so I'm not a particular expert on `matplotlib`.)

Install it with `pip install matplotlib`. If we have several Pythons and we're not good at this, and we don't know where we have `pip` and so on, it's easiest to just install it in Jupiter. We're going to use it inside Jupyter anyway, so there's nothing wrong if it might only be installed in the environment we're using for Jupyter.

In the cell you write

```
...
```

```
%pip install matplotlib
```

```
...
```

and execute it, and it will. If you don't have matplotlib yet, you will then need to restart Python using Kernel/Restart.

Now import `pyplot` from the `matplotlib` module under the name `plt` (to reduce typing).

```
import matplotlib.pyplot as plt
```

One last thing:

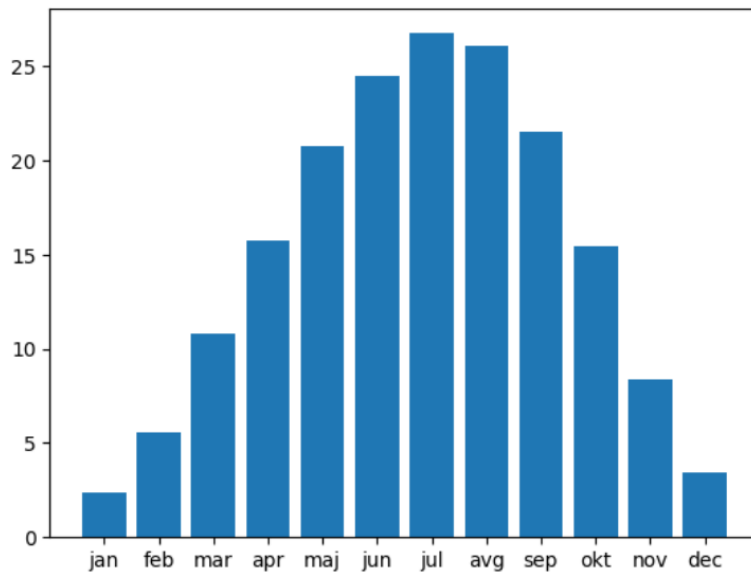
```
%matplotlib inline
```

These are the magic words that tell matplotlib to display the images in the notebook. At least one alternative that should probably work is `%matplotlib qt`: the graphs will be in a separate window. `inline` is more practical and, at least for me, seems to be the default.

Then open the matplotlib web page and choose a graph type. For temperatures, probably the most suitable will be bars, [https://matplotlib.org/stable/plot_types/basic/bar.html#sphx-glr-plot-types-basic-bar-py].

```
plt.bar(calendar.month_abbr[1:], monthly)
```

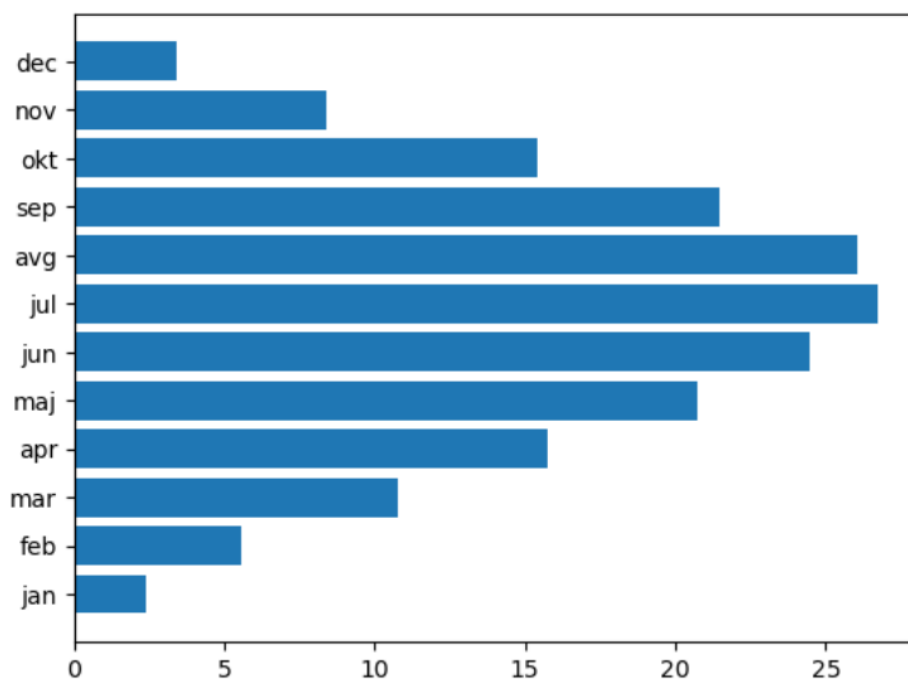
<BarContainer object of 12 artists>



Since I know that nobody will ask whether this graph can also be drawn horizontally, because that is stupid, I will do it on my own initiative.

```
plt.barh(calendar.month_abbr[1:], monthly)
```

<BarContainer object of 12 artists>

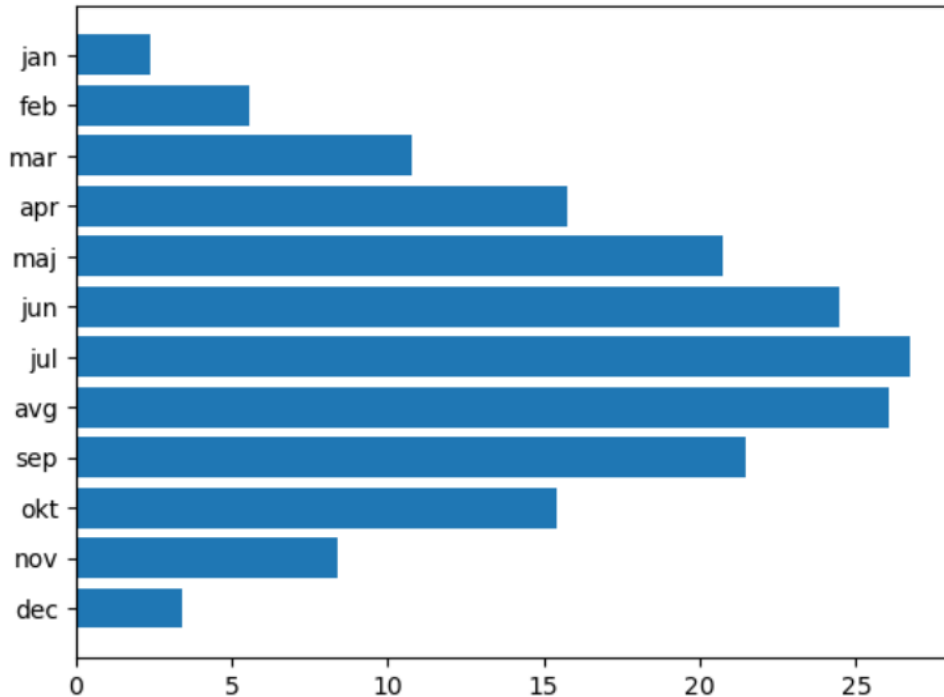


To make it look a little less awkward, let's reverse the `y` axis.

```
plt.axes().invert_yaxis()  
plt.barh(calendar.month_abbr[1:], monthly)
```

[33]:

<BarContainer object of 12 artists>

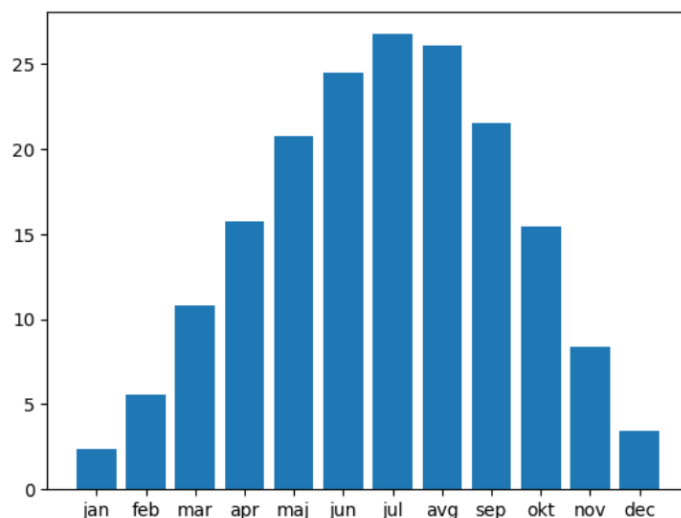


plt.axes()?! A person who is not familiar with matplotlib would naturally think that these are the axes of a graph. In fact, the axes are what "draw" the graph. The very first thing we did is actually just a shorthand for

```
ax = plt.axes()  
ax.bar(calendar.month_abbr[1:], monthly)
```

[34]:

<BarContainer object of 12 artists>

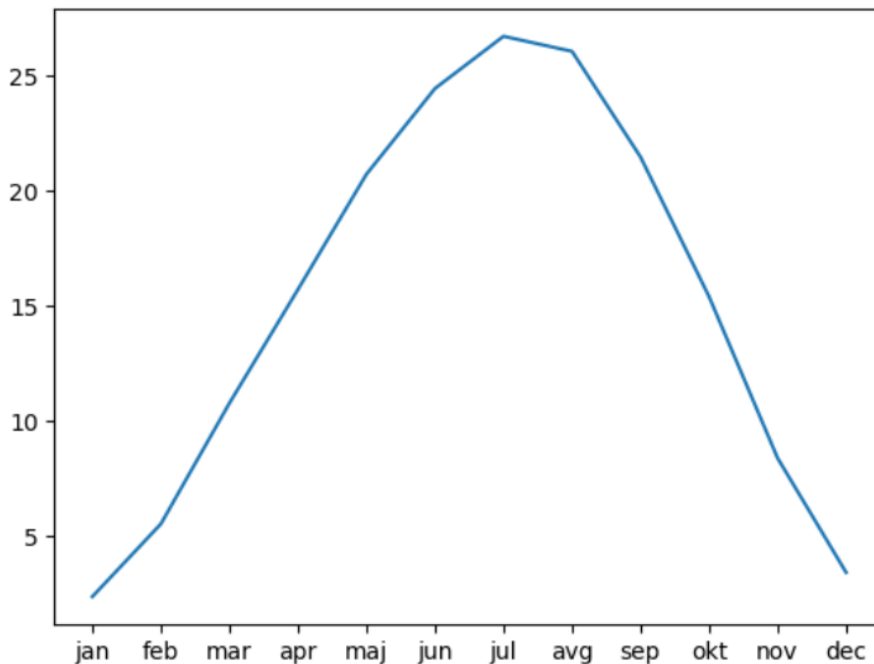


Let's not go further in that direction - at least not until we have to. Let's look at this instead: if we want to draw a curve instead of columns, we call

```
plt.plot(calendar.month_abbr[1:], monthly)
```

[35]:

[<matplotlib.lines.Line2D at 0x114a53e90>]



But no one is stopping us from drawing both.

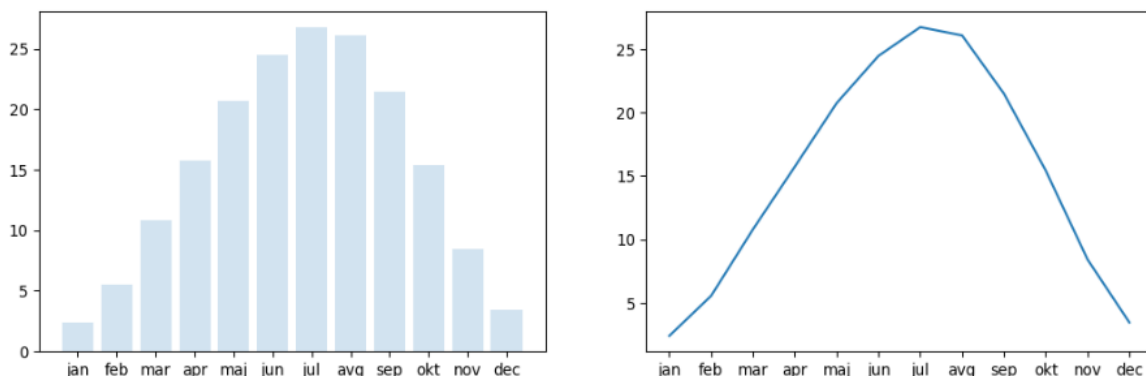
```
plt.figure(figsize=(13, 4), dpi=96)

ax = plt.subplot(1, 2, 1)
ax.bar(calendar.month_abbr[1:], monthly, alpha=0.2)

ax = plt.subplot(122)
ax.plot(calendar.month_abbr[1:], monthly)
```

[36]:

[<matplotlib.lines.Line2D at 0x126d3bf50>]



Oh, um, no, I didn't mean that ... I meant the same picture. But, anyway, while we're here - what is it?!

I used `plt.figure(figsize=(13, 4), dpi=96)` to say that I wanted a 13x4 inch figure (because inches, i.e. 2.54 centimetres, is the most standard unit of length, ever since all units switched to the decimal

system, in Europe, say, sometime around the time of the French Revolution), and that I wanted a resolution of 96 pixels per inch. In short, an image of

```
13 * 96, 4 * 96
```

pixels (pixels, in local terms). Why not give the resolution in pixels? Why in inches? If not centimetres? It is one thing what happens when we save the image. If the program we load it into is anything clever, it will keep those dimensions. The image will be large

```
13 * 2.54, 4 * 2.54
```

centimetres. Second, the font size is given in the units we know from office software, points. As you may or may not know, 1 pt equals 1/72 of an inch; 12 pt is therefore 12/72 of an inch, so $12/72 * 2.54$ cm, that is, just under half an inch.

```
12 / 72 * 2.54
```

Since the font is given in inches, we need to give the resolution of the image in dots per inch to determine how many dots (say on the screen) high the letter is.

We continued with

```
ax = plt.subplot(1, 2, 1)
```

```
ax.bar(calendar.month_abbr[1:], monthly, alpha=0.2)
```

```
ax = plt.subplot(122)
```

```
ax.plot(calendar.month_abbr[1:], monthly)
```

axes will be the "axes" that draw the graphs. With `plt.subplot(1, 2, 1)` we say that we would like to place the images in a grid with 1 row and 2 columns, and that we would now like to talk about the first of the images in this grid. Therefore 1, 2, 1. Then we draw the columns in this image. `alpha=0.2` makes the columns a bit more transparent, brighter.

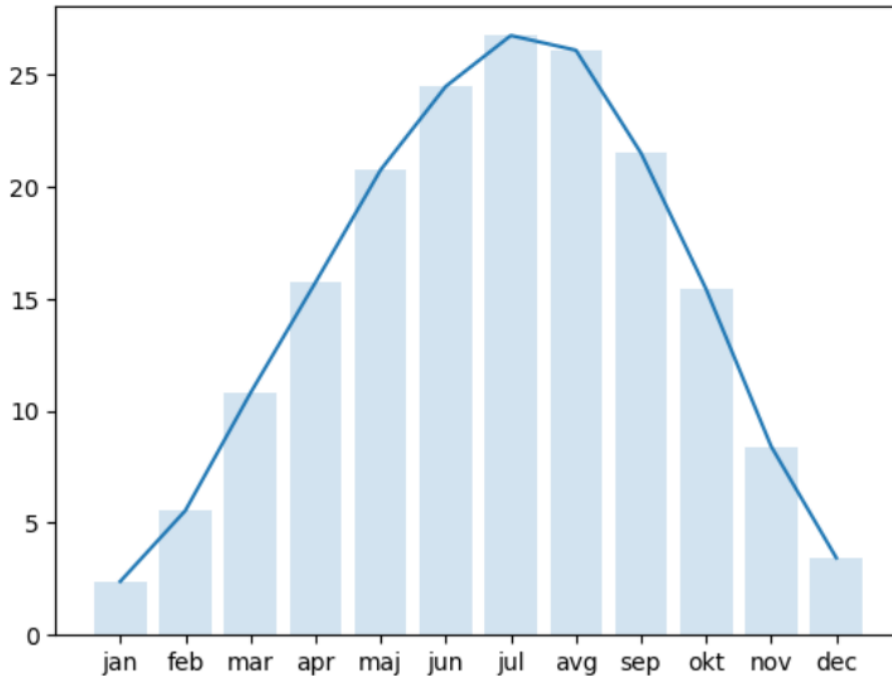
Then we say that we would draw the second image in this grid with 1 row 2 columns. Because we are lazy, we write 122 without commas. And then we draw the second picture.

But, as I said, I would like both in the same picture. This is simpler, of course. :)

```
plt.bar(calendar.month_abbr[1:], monthly, alpha=0.2)
plt.plot(calendar.month_abbr[1:], monthly)
```

[37]:

[<matplotlib.lines.Line2D at 0x126e73f50>]



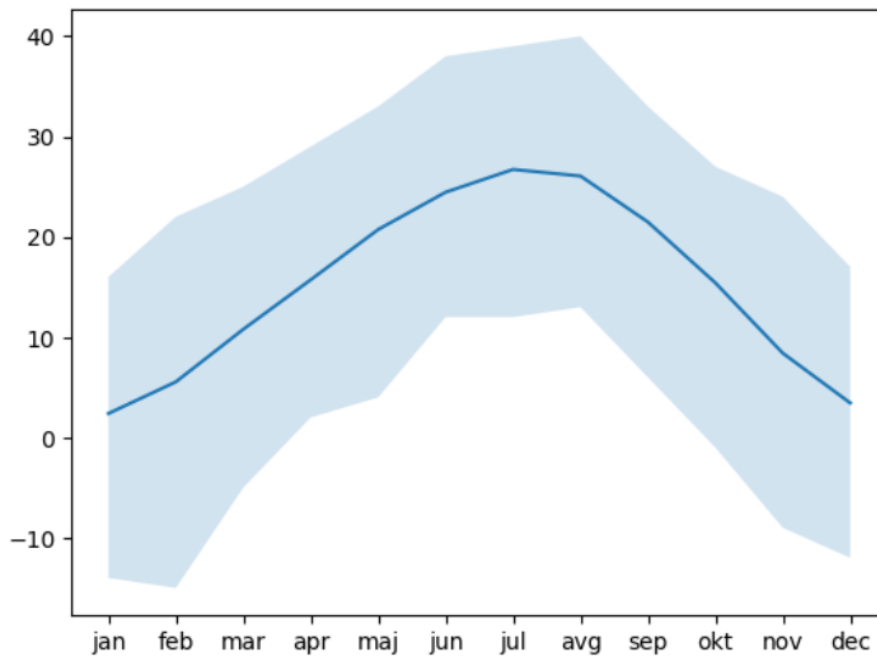
Why would anyone want that? I don't know. This picture shows the same thing twice. Edward Tufte would be turning in his grave if (a) he saw it and (b) he were no longer alive, but fortunately he is. (And if you ever get your hands on one of his books, just flick through it!)

It would be more appropriate to plot the maximum and minimum monthly temperature and the average. The graph we need is called `fill_between` and in addition to the x-axis data, we give two y-axis data - the upper and lower bounds. Over this, of course, we plot the average.

```
plt.fill_between(calendar.month_abbr[1:], monthly_min, monthly_max, alpha=0.2)
plt.plot(calendar.month_abbr[1:], monthly)
```

[38]:

[<matplotlib.lines.Line2D at 0x126f47d10>]



If you want nicer pictures, install `seaborn`: do `%pip install seaborn`, restart Python (Kernel / Restart), import it and set its theme.

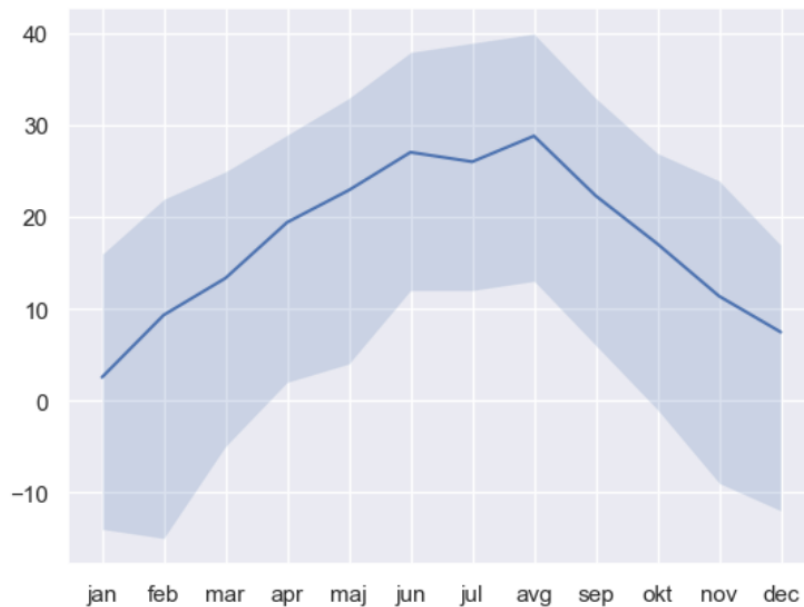
```
import seaborn

seaborn.set_theme()
```

```
plt.fill_between(calendar.month_abbr[1:], monthly_min, monthly_max, alpha=0.2)
plt.plot(calendar.month_abbr[1:], monthly)
```

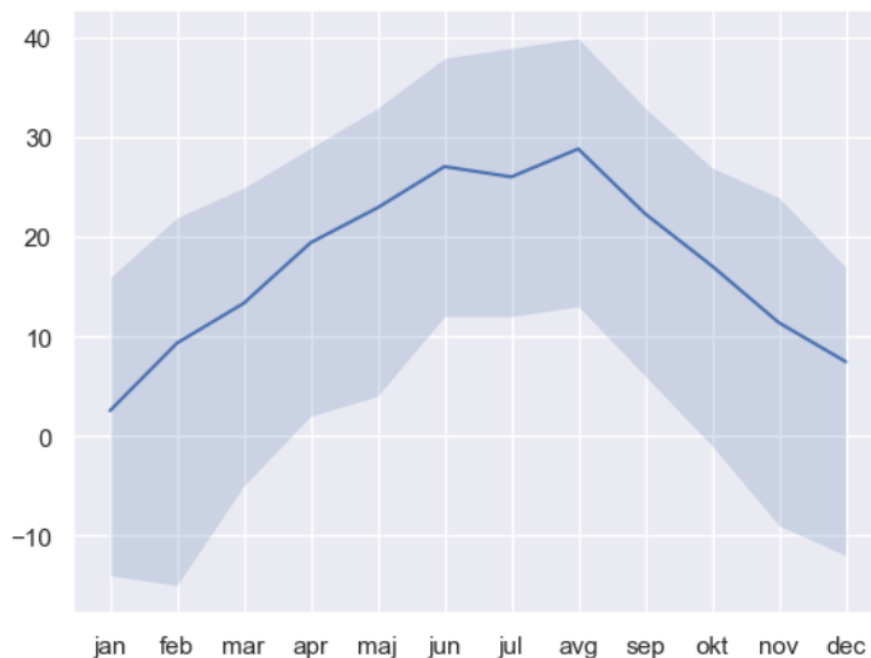
[122]:

[<matplotlib.lines.Line2D at 0x136703790>]



Save? Let's go! In png? Or svg? Maybe pfd? Right, pdf.

```
plt.fill_between(calendar.month_abbr[1:], monthly_min, monthly_max, alpha=0.2)
plt.plot(calendar.month_abbr[1:], monthly)
plt.savefig("mesečna-povprecja.pdf")
```



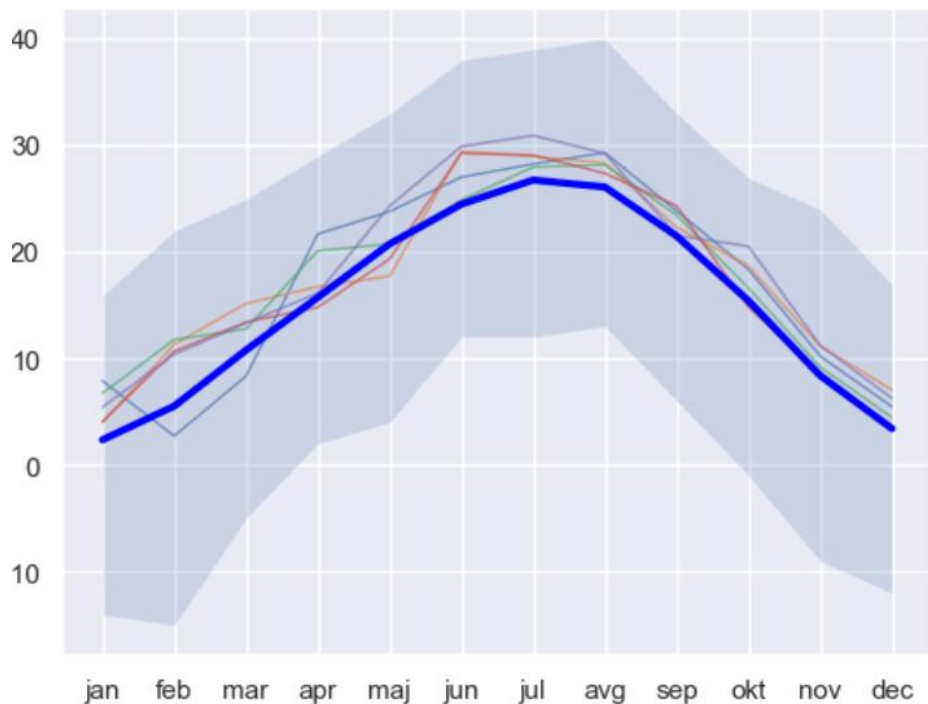
Different formats will work; at least .png will work everywhere, but the others depend on how your Jupyter is put together.

Would you add the average monthly temperatures for the last ten years to this graph?

```
months = calendar.month_abbr[1:]

plt.fill_between(months, monthly_min, monthly_max, alpha=0.2)
for year in range(118, 123):
    plt.plot(months, np.nanmean(temp[year], axis=1), linewidth=1, alpha=0.7)
plt.plot(months, monthly, linewidth=3, color="blue")
```

[<matplotlib.lines.Line2D at 0x12f7e3050>]

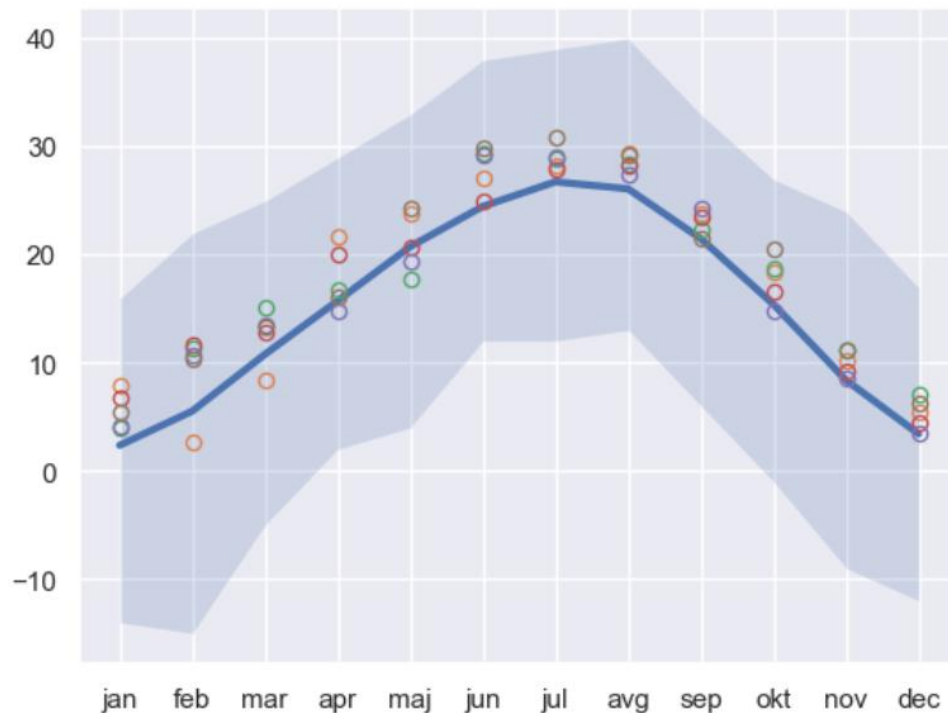


We are playing a little game: we have narrowed and lightened the lines for individual years, thickened the average and explicitly asked for it to be blue.

Not very convincing. Perhaps we would have preferred to replace the lines with circles?

```
names = calendar.month_abbr[1:]

plt.fill_between(names, monthly_min, monthly_max, alpha=0.2)
plt.plot(names, monthly, linewidth=3)
for year in range(118, 123):
    plt.plot(names, np.nanmean(temp[year], axis=1),
             linestyle='none', marker="o", markerfacecolor="none")
```



If we are interested in looking at something by month - maybe a moustache?

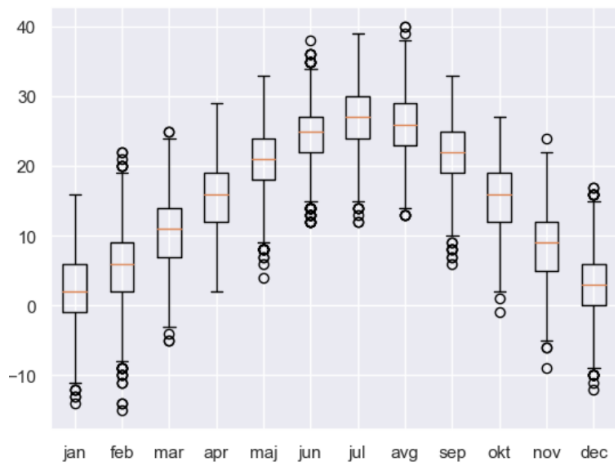
Let's make a list of tables: each element will contain all the temperatures measured in that month, in any year.

```
months = []
for month in range(12):
    x = temp[:, month, :].flatten()
    months.append(x[~np.isnan(x)])
```

With `temp[:, month, :]` we say that for the given month we would like all "rows" and "columns" - all years and days. The table will be two-dimensional - the first index will be the year, the second the day. With `flatten`, we push it into a single dimension. With `x[~np.isnan(x)]`, we fix only those elements that are not nan - that is, only known measurements.

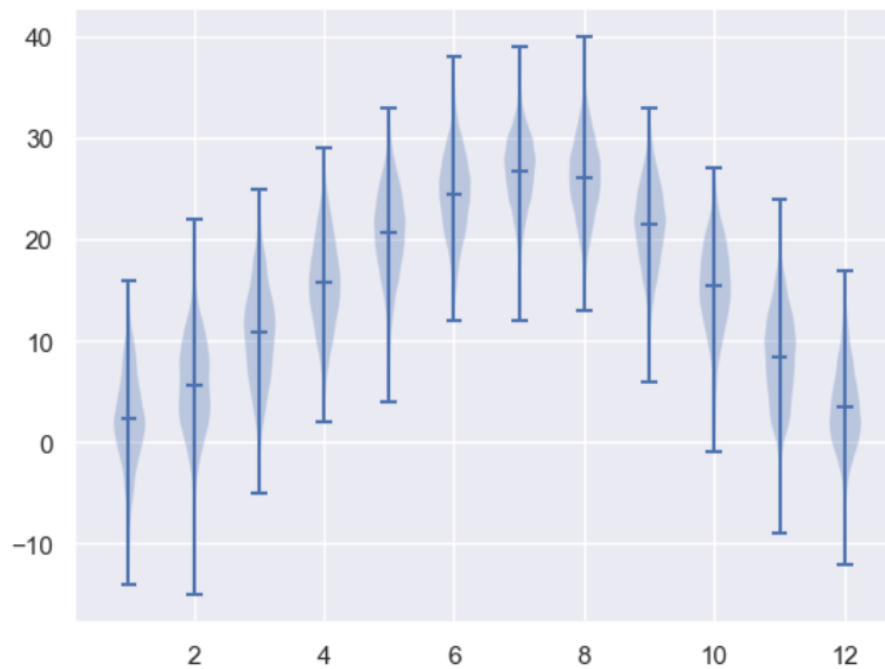
```
fig = plt.boxplot(months, labels=names)
```

since the function returns a bunch of things, we store them in `fig` so they don't get printed out



A similar thing, violins, show distributions.

```
_ = plt.violinplot(months, showmeans=True)
```



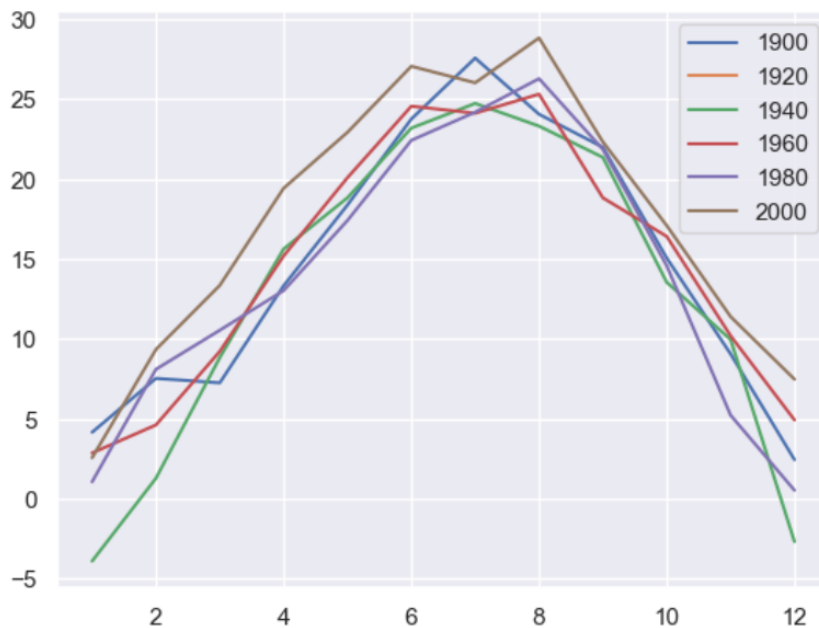
How about this? Let's plot average monthly temperatures by decade.

```
plt.plot(np.arange(1900, 2024), np.nanmean(temp, axis=(1, 2)))
```

/var/folders/2y/4j70c4q56811j41b6g1r0fk00000gn/T/ipykernel_69438/2795102103.py:1: RuntimeWarning: Mean of empty slice

```
plt.plot(np.arange(1900, 2024), np.nanmean(temp, axis=(1, 2)))
```

```
[<matplotlib.lines.Line2D at 0x146c2bbd0>]
```



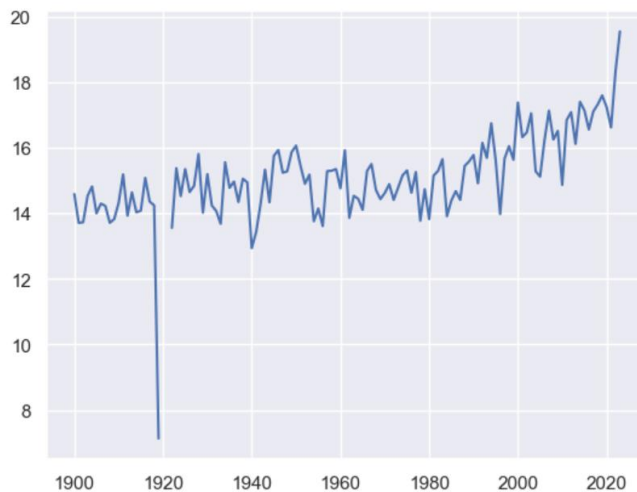
But this is quite interesting: it shows that the temperature is rising over the years. Who would have thought?

But here's how the average monthly temperature varies by year.

```
: plt.plot(np.arange(1900, 2024), np.nanmean(temp, axis=(1, 2)))
```

```
/var/folders/2y/4j70c4q56811j41b6g1r0fk00000gn/T/ipykernel_69438/2795102103.py:1: RuntimeWarning: Mean of empty slice
plt.plot(np.arange(1900, 2024), np.nanmean(temp, axis=(1, 2)))
```

```
: [<matplotlib.lines.Line2D at 0x146c2bbd0>]
```



We will stop here. I hope that's enough for you to see that serious graphs are not drawn with Excel. Matplotlib is probably the most serious graphing tool in science, next to R.

What we have seen here is nothing of the sort. We haven't scratched the surface yet. Take a walk through the gallery and you will see that there is probably everything you will ever need. And next to each image, there is a code that prints it out - a code that you can copy and adapt to your needs.