Introduction

August 19, 2024

1 Introduction

What are we aiming for? By the end of this course:

- You should be able to identifiy problems that can be solved with the help of a program.
- You should understand the basics of programming to design reasonably efficient and maintainable programs.
- You should know enough about Python's syntax to implement programs in it.
- You should be able to find help and access documentation.
- You should have an idea of some libraries that might be useful for your work.

1.1 The very Basics

Programming is telling a machine what to do. This has been done even before computers existed, e.g. for mechanical looms. For computers, programming can be done at different levels of abstraction. We could write the very instructions the hardware expects - but nobody does this. Instead, we will work with a high level language.

1.1.1 High Level Languages

There are many different high-level langages with different levels of abstraction availabel (C/C++, Java, Python, R, Haskell, …). They fall into two main groups:

- compiled: Commands are translated into harware instructions once before running. This usually runs faster.
- interpreted: Commands are translated at runtime. This is usually faster to develop, when you don't exactly know yet what you want (e.g. exploring data).

1.1.2 Python

- an interpreted langauge
- free software
- extensive documentation
- runs almost everywhere
- large collection of libraries
- large user base among scientists
- many nice tools that make life easier

1.2 Course Design

1.2.1 The Problem

Learning the basics of a language is "easy". Learning an additional langauge is very easy. The problems usually are:

- understanding the task clear enough
- splitting the task into managable chunks
- developing a clear idea of what you want the computer to do

I will try to teach you the basics and give a lot of tipps and hints on how to solve the three points above. I will also be happy to help when you struggle with a problem. But you really need as much hands on experience as possible.

1.2.2 Schedule, Slides and Exercises

Online at https://hsg.crbn.ch.

1.3 The Basics

1.3.1 Starting

There is different ways to start Python. For the first part, we will work with interactive python shell (in Spyder, Thonny or a Terminal).

Make sure everyone has a running python shell

1.3.2 Simple Calculations

You can use python as a simple calculator.

[1]:
```
21 + 21
```

[1]: 42

[2]:
```
50 - 8
```

[2]: 42

[3]:
```
14 * 3
```

[3]: 42

[4]:
```
756 / 18
```

[4]: 42.0

Note: we got a float (floating point number, a number containing a fractional part) above. Floats and ints (integers) are two different things in computers. They are stored completely differently.

Note 2: Floating point numbers have a limited precision.

[5]:
```
756 / 18 + 0.00000000000001
```

[5]:
```
42.00000000000001
```

[6]:
```
756 / 18 + 0.000000000000001   # one more 0
```

[6]:
```
42.0
```

Powers and operator precedence work as well

[7]:
```
6**2 + 6
```

[7]:
```
42
```

1.3.3 Errors

If we do something wrong, Python will tell us with an Error message

[8]:
```
6***2
```

```
  Cell In[8], line 1
    6***2
       ^
SyntaxError: invalid syntax
```

If you get an error, make sure to **really look at the message**. Python tries to help you figure out what's wrong. It tells you:

- in which file and on what line the error occured
- what this line looks like
- where in the line it thinks the error is
- what type of error it is, including a short explanation

1.3.4 Variables

Variables are one of the core concepts. Similar to what you are used to in math they represent a "value" and can be used in a formulas.

[9]:
```
value = 250
interest_rate = 2
years = 3
value*(1+interest_rate/100)**years
```

[9]:
```
265.302
```

It's easy now to repeat the calculation with a different value of the variables

[10]:
```
interest_rate = .5
years = 10
value*(1+interest_rate/100)**years
```

[10]: 262.7850330101974

You can also assign the result of a calculation to a variable. The calculation can even include this or another variable - as long as they already exist.

[11]:
```
value = value + 1000
result = value*(1+interest_rate/100)**years
result
```

[11]: 1313.925165050987

Naming Please use speaking variable names! (but try to keep it simple)

`(i*0.01+1)**y*v`

1.3.5 Scripts

You might want to repeat this calculation tomorrow. If you do not want to type everything in again, you want to write a script.

- Open a text file in your favourite editor
- Copy the lines into the file and save as `my_script.py`
- run `python3 my_script.py`

You will not see any output. The most important difference between scripts and interactive sessions is, that scripts only print output if you explicitly tell them so.

[12]:
```
value = 250
interest_rate = 2
years = 3
result = value*(1+interest_rate/100)**years
print(result)
```

265.302

1.3.6 Basic Types

So far we have seen two different types of numbers, ints and floats. You can ask python the type of a value or a variable with `type`.

[13]:
```
type(value)
```

[13]: int

[14]:
```
type(result)
```

[14]: float

There are many more types, but to get started I will just introduce the most basic ones here.

Text You probably will want to work with text at some point. Text is stored in Strings.

[15]:
```
print("The compound interest is:", result - value)
```

```
The compound interest is: 15.302000000000021
```

(Note the limited precision of floats)

[16]:
```
output_text = "The compound interest is:"
print(output_text, result - value)
```

```
The compound interest is: 15.302000000000021
```

[17]:
```
type(output_text)
```

[17]:
```
str
```

Lists We will soon start to work with more then one value at once. There most basic type for this is lists. They are created with `[]`.

[18]:
```
interest_rates = [.5,1,2]
print(interest_rates)
```

```
[0.5, 1, 2]
```

[19]:
```
type(interest_rates)
```

[19]:
```
list
```

You can access the elements with [] – Warning: indexing starts at 0!

[20]:
```
interest_rates[0]
```

[20]:
```
0.5
```

[21]:
```
print(value*(1+interest_rates[0]/100)**years)
print(value*(1+interest_rates[1]/100)**years)
print(value*(1+interest_rates[2]/100)**years)
```

```
253.7687812499999
257.57525000000004
265.302
```

(This code is is not really nice, but we will discuss this later)

You can assign each entry of the list to a different variable in one go (called unpacking)

[22]:
```
low, medium, high = interest_rates
print("low:", low, "-- high:", high)
```

```
low: 0.5 -- high: 2
```

(Unpacking is not so useful here, but will be a very common pattern later.)

If you want to know how many elements are in a list, use len()

[23]:
```
len(interest_rates)
```

[23]:
```
3
```

Accessing elements, unpacking and counting elements works for all data types containing elements too. Another example of suche a type are strings:

[24]:
```
print(len(output_text), output_text[0])
```

```
25 T
```

Booleans Booleans are either `True` or `False` and can be used in logical operations. We will talk about this more later.

[25]:
```
my_bool = value > 1000
my_bool
```

[25]:
```
False
```

[26]:
```
type(my_bool)
```

[26]:
```
bool
```

1.3.7 Calling Functions

We've seen the `print()`- and the `type()`- Function already.

[27]:
```
print(result)
```

```
265.302
```

There are many others, and you will soon be defining your own funtions. They are the main building-blocks of programs. The syntax to call a function is with round brackets containing the argument(s).

[28]:
```
round(result, 2)
```

[28]:
```
265.3
```

Calling a function tells python to run through the code contained in the function definition. So in a first approximation, a function is a block of code, stored somewhere else and using its own variables.

Remember: You call a function with () you access elements in a collection with []

[29]:
```
interest_rates()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[29], line 1
----> 1 interest_rates()

TypeError: 'list' object is not callable
```

[ ]:
```
round[0]
```

Some functions give back values we can store

[30]:
```
rounded_result = round(result)
```

[31]:
```
print(rounded_result)
```

```
265
```

others don't

[32]:
```
p = print(result)
```

```
265.302
```

[33]:
```
print(p)
```

```
None
```

`None` is the value indicating, that print() did not return anything

On the other hand some functions don't have input.

[34]:
```
copyright()
```

```
Copyright (c) 2001-2023 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

[35]:
```
#exit()
```

1.3.8 Modules

Python has a bunch of functions like `round` and `print` built-in. But the real power is all the additional collections of functions available. These collections are often called libraries or modules in Python.

You add them with the `import` statement.

[36]:
```
import numpy
```

Once imported, you can use `module.function` to use the new functions.

[37]:
```
numpy.sqrt(4)
```

[37]:
```
2.0
```

People who do not like to type so much, can use the following shorthand.

[38]:
```
import numpy as np
np.sqrt(9)
```

[38]:
```
3.0
```

There are other ways to load things, but for now this syntax is enough.

1.4 Hands On 1

1.4.1 Questions

- How do you import a module in python?
- How do you assign a value to a variable?
- What is the difference between `a = 42` and `a == 42`
- How do you call a function?
- How do you call a function from a module?
- How do you define a list?
- What is the difference between `res = thing(1)` and `res = thing[1]`?

1.4.2 Tasks

- repeat the things we did, type in the commands yourself and experiment with them.
- What does the following program print a = -1 b = 2 c, d = b, a print(c > 0)
- Implement a simple formula from your own domain as a python script.
- Calculate the mean of the following numbers: `2,3,5,7,11,13`
  - manually by tipping the calculation
  - using a list and `numpy.mean`

2 Looking at Refugee Data

As a toy study for this tutorial, we will look at the development of the refugee populations in the different EU countries. I downloaded the data files from the World Bank. We will work with these

later. But to make things easier for the start, I created a few seperate files in a format directly understood by Python.

[39]:
```
#!cat data/refugees-europe_0.txt
```

[40]:
```
#!cat data/countries-europe.csv
```

If the data has the optimal format, we can load it simply by calling `numpy.loadtxt`. This returns an array with the data from the file, skipping rows starting with `#`.

[41]:
```
import numpy as np
refugees = np.loadtxt("data/refugees-europe_0.txt")
#print(refugees)
```

The output looks like we got a list of lists. However an array is a different type, introduced by numpy. It is heavily optimised to work with data and used by many scientific modules.

[42]:
```
type(refugees)
```

[42]:
```
numpy.ndarray
```

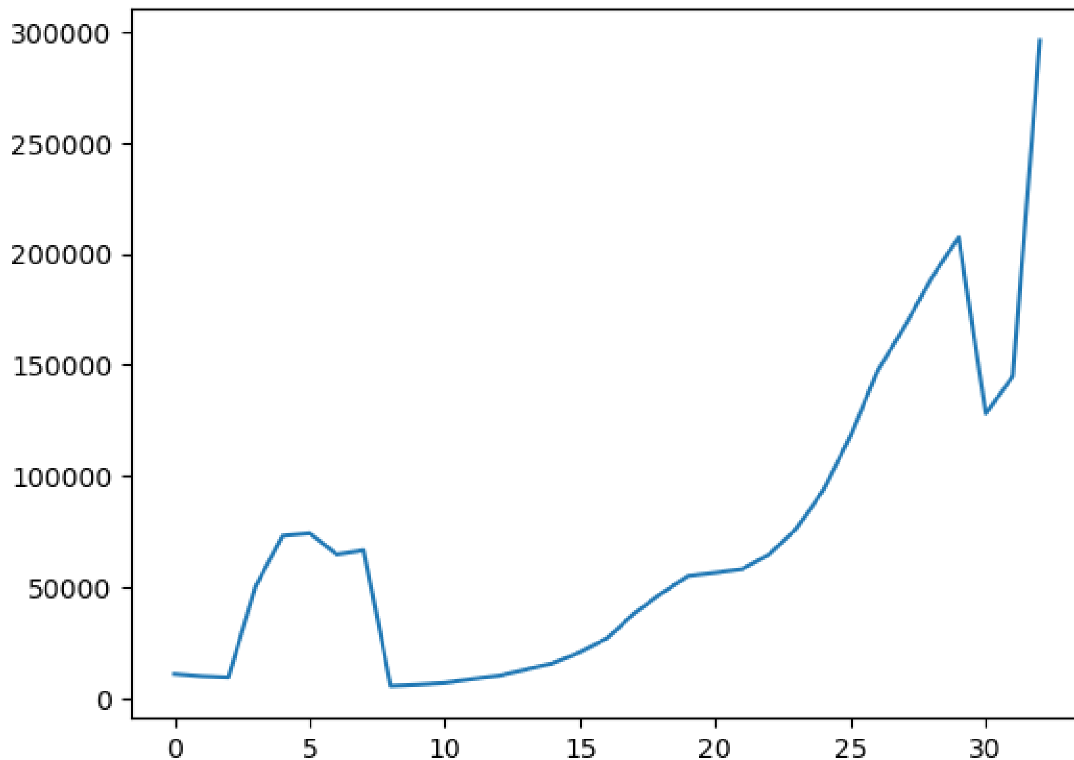
Like with lists we can access a single element (a full row in our case) with square brackets

[43]:
```
refugees[14] # Italy
```

[43]:
```
array([ 10816.,   9796.,   9242.,  50266.,  73058.,  74298.,  64712.,
        66601.,   5440.,   6004.,   6813.,   8541.,  10031.,  12812.,
        15668.,  20674.,  26870.,  38061.,  47059.,  54962.,  56402.,
        58066.,  64782.,  76248.,  93712., 118036., 147362., 167330.,
       189227., 207602., 128033., 144862., 296181.])
```

Of course we want picutes…

[44]:
```
import matplotlib.pyplot as plt # the default plotting library
plt.plot(refugees[14])
plt.show()
```



2.1 Intermezzo: Objects

An object is a programming concept, grouping data and code together. We will discuss why this is a good idea and how to do this yourself in more detail on Wednesday (object oriented programming).

Here we just want to understand how to use existing objects in Python. Objects have attributes:

- data attributes (variables, accessed without brackets)
- methode attributes (functions, called with round brackets)

We access the attributes with the syntax `object.attribute`. If this seems familiar, you are right. In python *everything is an object*. In the code above, the `plot` function is a methode attribute of the imported `matplotlib.pyplot` module-object.

The easiest way to get an overview of all available attributes is to use tab-completion in an interactive session.

Some example of more things that are objects

[45]:
```
# numbers
num = 42
num.numerator
```

[45]: 42

[46]:
```
# functions
np.loadtxt.__str__()
```

[46]: `'<function loadtxt at 0x7fad5d7d49a0>'`

[47]:
```
print(np.loadtxt)
```

```
<function loadtxt at 0x7fad5d7d49a0>
```

Knowing that everything is an object, we can have a second look at variables as well.

All objects exist somewhere in the memory of our computer (cf. memory-address above), and variables are just labels we attach to this objects ("pointers to these memory locations").

This is important to understand what happens here:

[48]:
```
my_list = [1,2,3]
my_other_list = my_list
my_other_list[1] = 0
print(my_list)
```

```
[1, 0, 3]
```

This does not happen with numbers or strings, as you can not change the content of a these objects (they are immutable).

[49]:
```
my_string = "Hello World"
my_other_string = my_string
my_other_string[0] = "h"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[49], line 3
      1 my_string = "Hello World"
      2 my_other_string = my_string
----> 3 my_other_string[0] = "h"

TypeError: 'str' object does not support item assignment
```

So when ever you change such a variable, the label must point to some new location in memory.

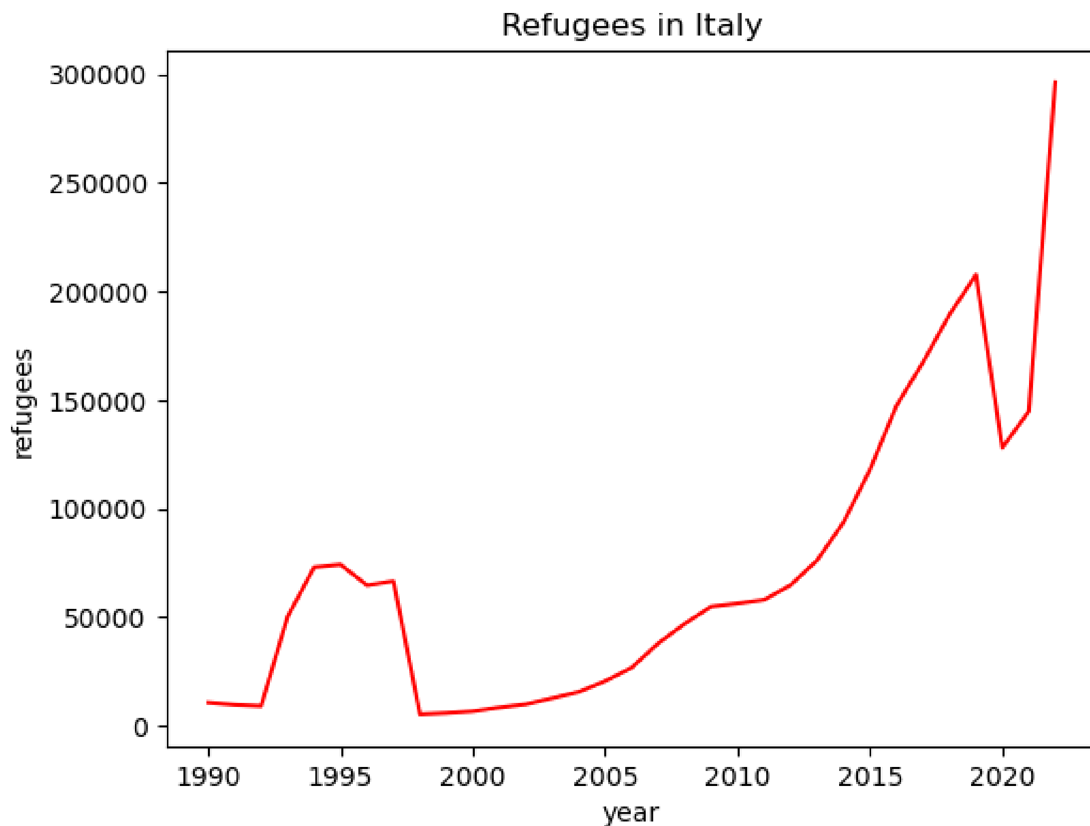
Note: In Thonny you can make this visible by showing *Variables* and *Heap* (both in the *View*-Menu)

2.2 Nice Plots

I hope you all would recject a plot like the one above, if you got that from a student. Let's turn this into a useful plot.

[50]:
```
years =␣
 ↪[1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,
figure = plt.figure()
```

```
axes = figure.subplots()
axes.plot(years, refugees[14], color="r")
axes.set_xlabel("year")
axes.set_ylabel("refugees")
axes.set_title("Refugees in Italy")
#axes.set_ylim(0,max(refugees[14])*1.05)
plt.show()
```



Let's discuss this in details

- we create a list with all the years in our data by copy-paste from the data-file (the colum-labels are a comment, because we do not want to mix different types of data)
- we create a new figure-object
- we add an axes-object to that figure that will hold our plot (will look at the details later)
- we call the plot methode from our axes object it works identical to the one we called before
  - if passed a singel list these are the y-values and they will be continously numbered for x
  - if passed two arguments these are x- and y-values
  - you can pass many more arguments to configure the line (see the documentation)
- We then use three methodes from the axes object to set x- and y-labels as well as a title of this object. (and maybe the range)
- Finally we call the show-function from the pyplot module to display the plot we just created.

2.2.1 Finding Help

You find a lot of documentation online, but normally the documentation is even availabel directly in Python.

[51]:
```
#help(axes.plot)
```

[52]:
```
#axes.plot?  # ipython
```

[53]:
```
#!pydoc3 matplotlib.pyplot.plot
```

2.2.2 Comments

You might want to add comments to your code as a note to your future self or other people that might need to understand your program. There are two options:

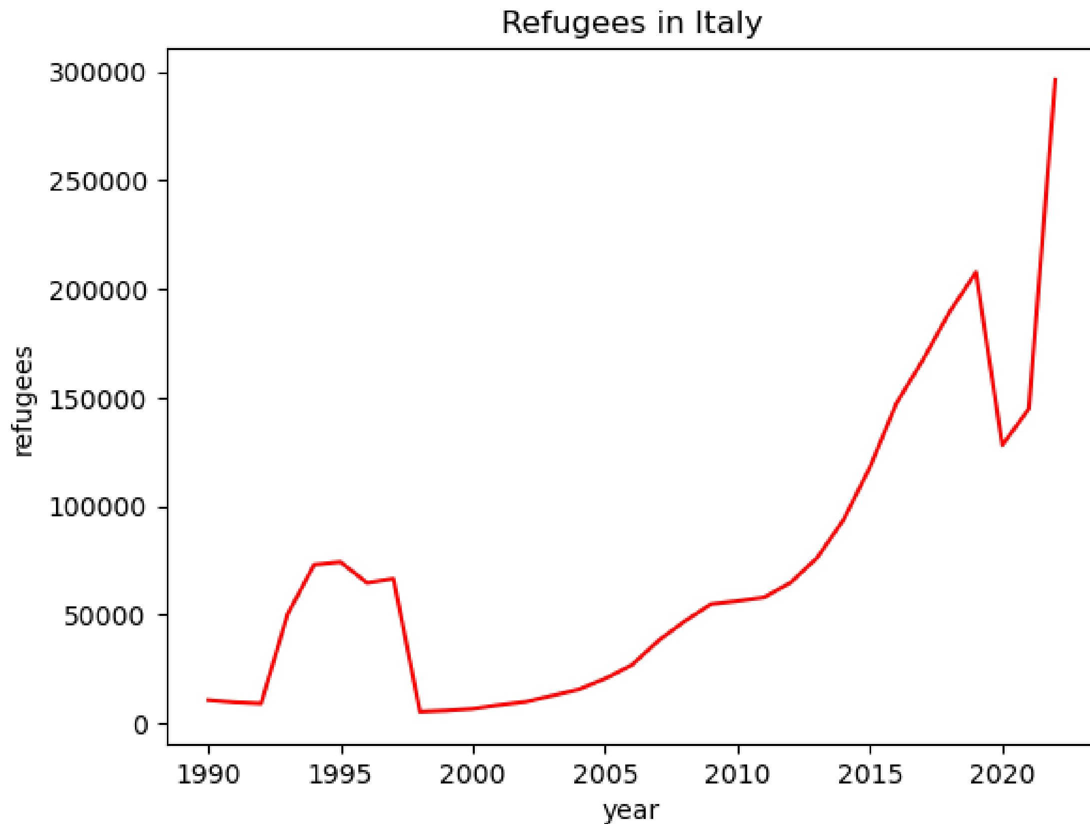
[54]:
```
# a single-line comment is introduced with a hash-sign
```

[55]:
```
"""
a multi-line string is enclosed in triple-quotes
and can be used as a comment
"""
```

[55]:
```
'\na multi-line string is enclosed in triple-quotes\nand can be used as a
comment\n'
```

(This acutally creates a multi-line string-object, that you could asign to a variable)

[56]:
```
""" Refugees in Italy

Display the number of refugees over time in Italy
"""
import numpy as np
import matplotlib.pyplot as plt
# define data variables
years =␣
 ↪[1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,
refugees = np.loadtxt("data/refugees-europe_0.txt") # data from World Bank
# create axes object
figure = plt.figure()
axes = figure.subplots()
# plot the data
axes.plot(years, refugees[14], color="r")
# add labels and title
axes.set_xlabel("year")
axes.set_ylabel("refugees")
axes.set_title("Refugees in Italy")
# display the plot
plt.show()
```



2.3 A Closer Look at Arrays

So far we have just loaded data from a file into an array and then accessed individual rows from it. But arrays are a lot more powerful.

To look closer at arrays, we first create a simple test-array:

[57]:
```
test_array = np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(test_array)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
```

Make sure you understand the meaning of round and square brackets

Note: We passed the array-Function a list containing lists here. Lists can contain all kind of things, so you can create arbitrary complex constructs. (but please aviod that :-)

2.3.1 Accessing Elements

As we have seen, we can access elements with `[]` as well. (Remember: indexing starts at 0!)

[58]:
```
first_row = test_array[0]
print(first_row)
```

```
[1 2 3 4 5]
```

[59]:
```
print(first_row[1])
```

```
2
```

You can also put the different dimensions in one square bracket, separated by comma. As with the separate bracktes, the order is row, column.

[60]:
```
print(test_array[0,1])
```

```
2
```

You might want to choose only a slice of an array. For this you can use the syntax `lower:upper`.

[61]:
```
print(test_array[1, 2:4])
```

```
[8 9]
```

Note: The upper bound is not included!

If you leave a boundary empty, it will select everything until the edge of the array

[62]:
```
print(test_array[:, :2])
```

```
[[1 2]
 [6 7]]
```

You can also index from the end using negativ indices.

[63]:
```
print(test_array[-1, -2:])
```

```
[ 9 10]
```

And you can even choose a stepsize

[64]:
```
print(test_array[0,1:4:2])
```

```
[2 4]
```

Such selections are called slices. You can store them in a new variable.

[65]:
```
my_slice = test_array[:, 2:4]
print(my_slice)
```

```
[[3 4]
 [8 9]]
```

And you can assign to them

[66]:
```
test_array[-1, -2:] = 11
print(test_array)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8 11 11]]
```

but be aware, a slice is not a separate array

[67]:
```
print(my_slice)
```

```
[[ 3  4]
 [ 8 11]]
```

[68]:
```
my_slice[:,:] = -1  # what happens without the [:,:]?
```

[69]:
```
print(my_slice)
```

```
[[-1 -1]
 [-1 -1]]
```

[70]:
```
print(test_array)
```

```
[[ 1  2 -1 -1  5]
 [ 6  7 -1 -1 11]]
```

Ohter Ways to Access Elements For arrays, there are two more useful ways to access elements.

- You can pass a list of indices to an array. Numpy will then build tuples from the different lists and access the corresponding elements.

[71]:
```
test_array[[1,0,1],[0,1,2]]
```

[71]:
```
array([ 6,  2, -1])
```

Doing this, numpy is quiet smart. You can define a new shape for the result and it expands missing values when possible.

[72]:
```
test_array[[[1,0],[1,0],[1,0]],[[0,0],[1,1],[4,4]]]
```

[72]:
```
array([[ 6,  1],
       [ 7,  2],
       [11,  5]])
```

[73]:
```
test_array[[1],[0,1,4]]
```

[73]:
```
array([ 6,  7, 11])
```

- You can pass an "array" of the same shape, containing booleans. This array then works as a mask. Numpy will then select all elements where the mask contains True

[74]:
```
mask = np.array([[True, False, True, False, True], [True, False, True, False,␣
↪True]])
test_array[mask]
```

[74]:
```
array([ 1, -1,  5,  6, -1, 11])
```

2.3.2 Attributes of Arrays

Of course arrays are objects too. They have many useful attributes. Type `test_array.` then press tab.

For example, you can ask the array for its shape or aggregate the values along a given axis.

[75]:
```
refugees.shape
```

[75]:
```
(28, 33)
```

[76]:
```
refugees.sum(axis=0) # total number of refugees in a given year
```

[76]:
```
array([1350375., 1382912., 2298412., 2409878., 2324491., 2260144.,
       2279092., 1972505., 1758335., 1798380., 1707299., 1743816.,
       1827288., 1784085., 1680736., 1491462., 1398611., 1364949.,
       1390299., 1414535., 1394187., 1346811., 1339344.,  976710.,
       1097130., 1333779., 1888718., 2288959., 2489547., 2724432.,
       2805406., 2996000., 7040942.])
```

[77]:
```
refugees.mean(axis=1) # mean number of refugees per country
```

[77]:
```
array([ 66310.72727273,  31423.84848485,  11802.78787879,  44364.12121212,
         4143.06060606,  15333.72727273,  43309.45454545,   1310.24242424,
        13929.27272727, 225212.12121212, 900264.78787879,  22927.39393939,
        10678.63636364,   7301.36363636,  71502.03030303,   1323.09090909,
         2674.24242424,   2142.84848485,   3962.87878788,  97492.21212121,
        35981.39393939,   2590.21212121,   4901.81818182,   3546.84848485,
         6578.51515152,  23990.48484848, 154282.54545455, 171312.33333333])
```

2.3.3 Math with Arrays

One of the major advantages of arrays is, that math involving arrays is done elementwise.

[78]:
```
test_array*2
```

[78]:
```
array([[ 2,  4, -2, -2, 10],
       [12, 14, -2, -2, 22]])
```

Doing the same with lists, would not work as expected.

[79]:
```
[1,2,3]*2
```

[79]:
```
[1, 2, 3, 1, 2, 3]
```

Question: Using arrays, how would you simplify the calculation of the compound interest for 3 different interest rates?

[80]:
```
value = 250
interest_rates = np.array([0.5, 1, 2])
years = 3
```

```
results = value*(1+interest_rates/100)**years
print(results)
```

```
[253.76878125 257.57525    265.302     ]
```

2.4 Hands On 2

2.4.1 Questions

- What does `plot([0, 0, 1, 1], [0, 1, 0, 1])` display?
- What kind of attributes does an object have in python? What is the difference when accessing them?
- Explain in your own words what this code prints and why: my_list = [1,2,3] my_other_list = my_list my_other_list[1] = 0 print(my_list)
- What methods are available to get help?
- How can you add a comment to python code?
- Slicing also works on lists and strings. Given `element = 'oxygen'` what do the following commands return:
  - `element[:4]`
  - `element[4:]`
  - `element[:]`
  - `element[-1]`
  - `element[-2]`
  - `element[1:-1]`
- What is the result of the two print statments in the following code: my_list = [1,2,3] my_array = np.array([1,2,3]) print(my_list * 2) print(my_array * 2)

2.4.2 Tasks

- Repeat the things we did, type in the commands yourself and experiment with them.
- Replace `refugees-europe_0.txt` with `refugees-europe_1.csv`, adapt the call to `loadtxt` according to the new input format.
- Get a general idea of the functions provided by numpy. (use tab-completion and the help-system)
- Plot the refugee population of Germany over the periode from 2000 unitl 2010.
- Use the `axes.set_ylim` and `array.max` to make sure the y axis of your plot starts at `0` and includes all data.
- Draw a horizontal line into your plot, indicating the mean.
- Use `numpy.diff` to plot the change in refugees instead of the absolut value.

3 Repeating Things

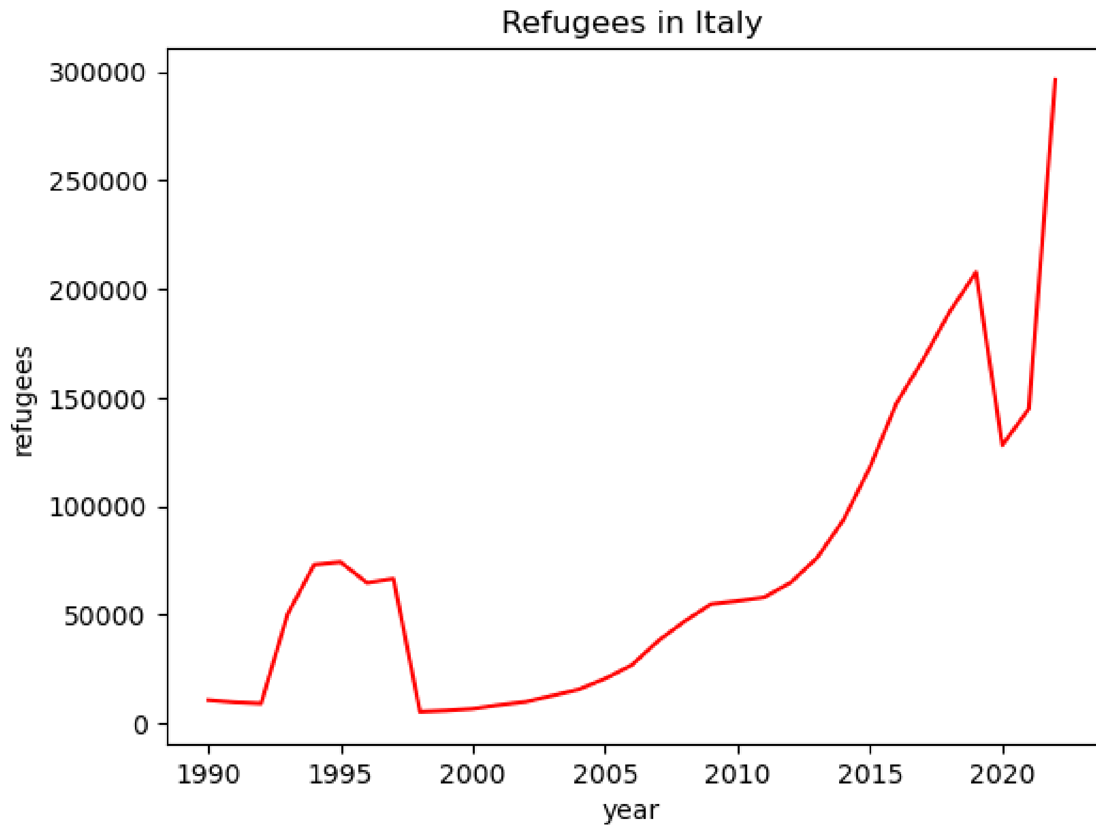
We do have many different datasets. And most likely we will want to include all of them in whatever analysis we run. Let's start with creating the plot above for several countries.

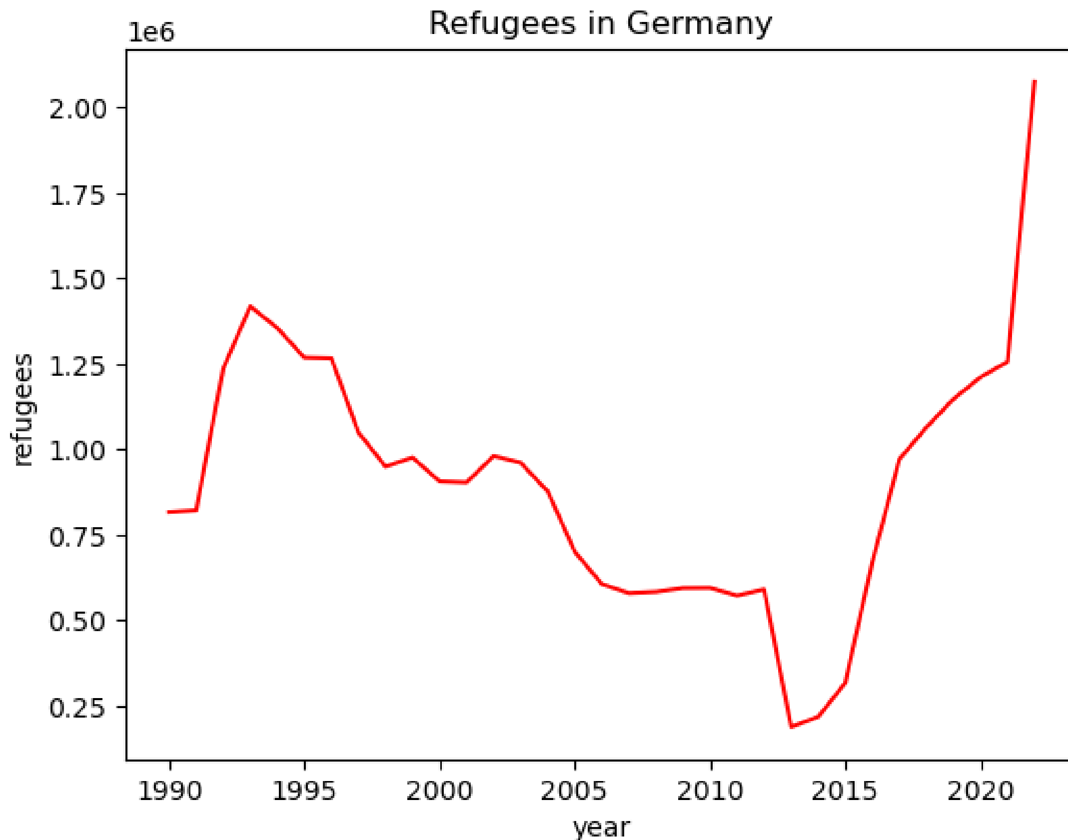
3.1 How not to do it

The first option is of course to copy-paste the code and adapt it. Let's just add Germany.

[81]:

```
import numpy as np
import matplotlib.pyplot as plt
# define data variables
years =␣
 ↪[1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,
refugees = np.loadtxt("data/refugees-europe_0.txt") # data from World Bank

# create axes object
figure = plt.figure()
axes = figure.subplots()
# plot the data
axes.plot(years, refugees[14], color="r")
# add labels and title
axes.set_xlabel("year")
axes.set_ylabel("refugees")
axes.set_title("Refugees in Italy")

# create 2nd axes object
figure2 = plt.figure()
axes2 = figure2.subplots()
# plot the data
axes2.plot(years, refugees[10], color="r")
# add labels and title
axes2.set_xlabel("year")
axes2.set_ylabel("refugees")
axes2.set_title("Refugees in Germany")
# display the plot
plt.show()
```


Refugees in Italy
refugees
300000
250000
200000
150000
100000
50000
0
1990
1995
2000
2005
2010
2015
2020
year



3.1.1 Why no to do this

- Are you looking forward to adding the other 26 countries?
- Once you are done, I'll tell you that I want the Axis labels properly capitalized.

3.2 For-Loops

To repeat parts of the code, all programming languages offer loops. The easiest to understand is a for-loop. A for-loop allows us to repeat a block of code for every item in a sequence.

If we want to ask for different types of cheese, instead of writing several almost identical print statements

[82]:
```
print("How are you on Tilsit?")
print("How are you on Gruyere?")
print("How are you on Emmental?")
```

```
How are you on Tilsit?
How are you on Gruyere?
How are you on Emmental?
```

we can write a for-loop.

[83]:
```
for cheese in ["Tilsit", "Gruyere", "Emmental"]:
    print("How are you on ", cheese, "?")
print("Not part of the loop.")
```

```
How are you on  Tilsit ?
How are you on  Gruyere ?
How are you on  Emmental ?
Not part of the loop.
```

Note: The code block is defined by the indentation.

[84]:
```
for cheese in ["Tilsit", "Gruyere", "Emmental"]:
    print("How are you on ", cheese, "?", sep="")
```

```
How are you on Tilsit?
How are you on Gruyere?
How are you on Emmental?
```

Iterating works for many different types of sequences

[85]:
```
for letter in "ABC":
    print(letter)
```

```
A
B
C
```

[86]:
```
total = 0
for number in np.array([1,2,3]):
    total = total + number
    print("adding", number)
print("total:", total)
```

```
adding 1
adding 2
adding 3
total: 6
```

see what's happening Sometimes it is really useful to visualise what happens. This is easiest using Thonny. We will look at the default debugger later in this course as well - and finally you could use this website:

`http://www.pythontutor.com`

(Unfortunately the website does not support numpy)

3.2.1 Useful helpers

Python has a few helper functions, that are especially useful when working with for-loops.

range([min=0], sup, [step=1]) Returns the numbers from min to sup (excluding sup) with step increments.

[87]:
```
list(range(2,10,3))
```

[87]:
```
[2, 5, 8]
```

So we could write the last loop as

[88]:
```
for number in range(1,4):
    print(number)
```

```
1
2
3
```

You do not need to use the loop variable inside the loop. This is also useful to run some code N-times.

[89]:
```
for n in range(5):
    print("ROMANI ITE DOMUM!")
```

```
ROMANI ITE DOMUM!
ROMANI ITE DOMUM!
ROMANI ITE DOMUM!
ROMANI ITE DOMUM!
ROMANI ITE DOMUM!
```

enumerate(sequence) Enumerate returns each item in a sequence together with its position (remember unpacking):

[90]:
```
for pos, value in enumerate("ABC"):
    print(pos, value)
```

```
0 A
1 B
2 C
```

zip(...) Zip allows to iterate over two (or more) lists (or other iterables) in parallel.

[91]:
```
for letter, number in zip("ABC", range(3)):
    print(letter, number)
```

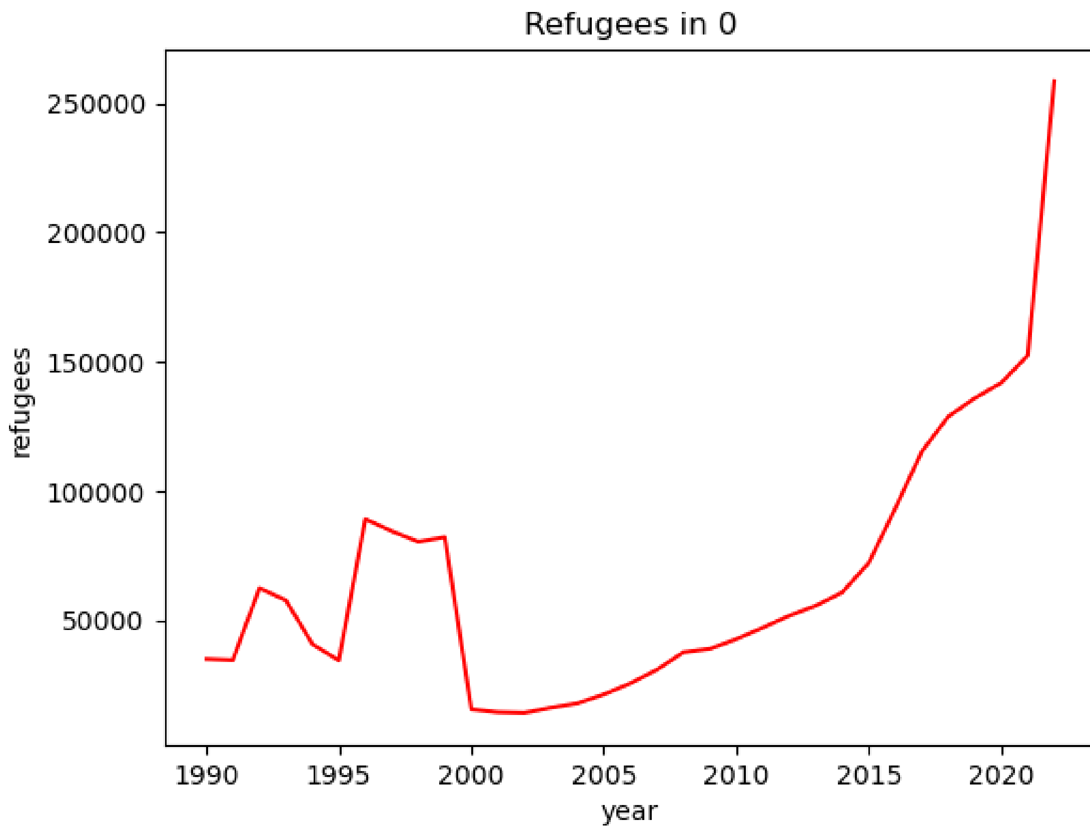
```
A 0
B 1
C 2
```

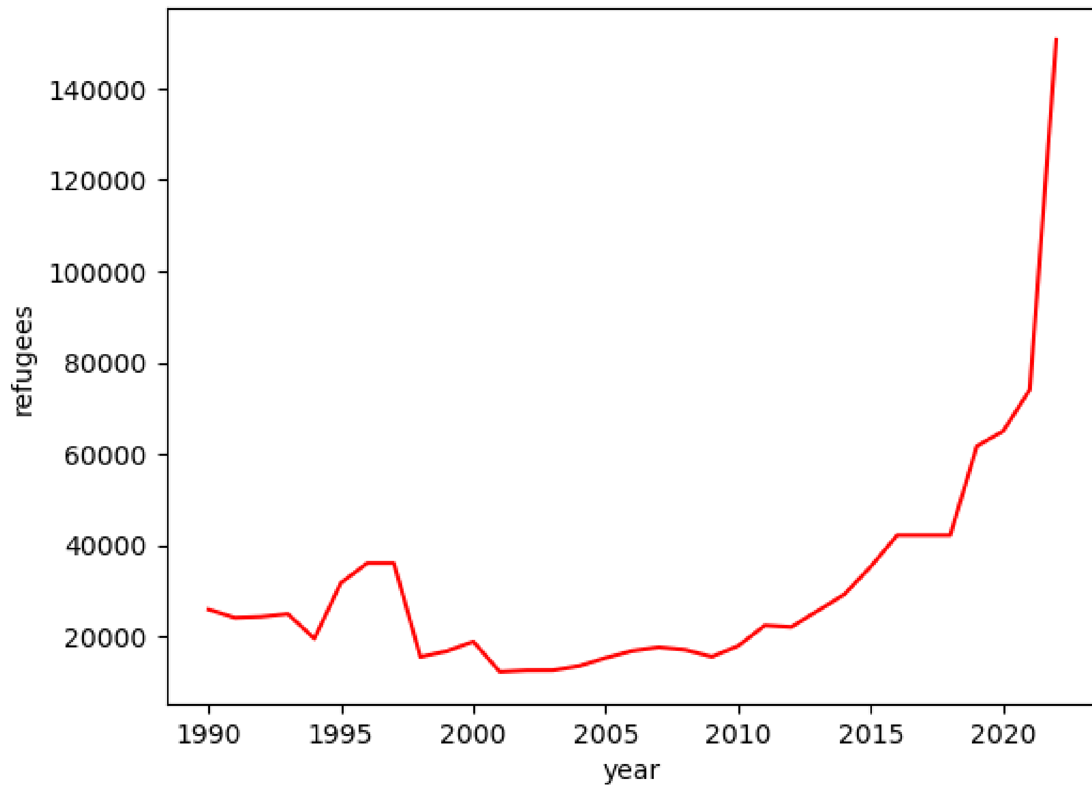
3.3 Plotting in a Loop

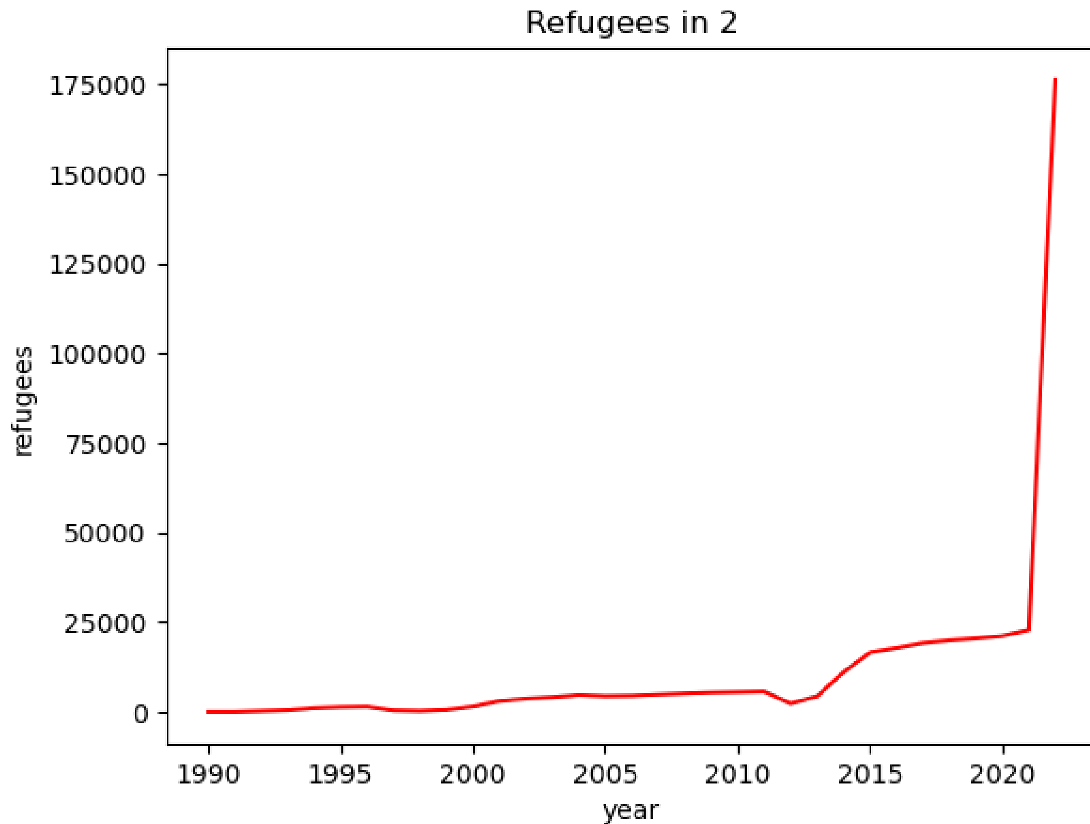
With this new knowledge of loops, let's rewirte the plotting code.

[92]:

```
import numpy as np
import matplotlib.pyplot as plt
# define data variables
years = range(1990,2023)
refugees = np.loadtxt("data/refugees-europe_0.txt") # data from World Bank

for row in range(3):
    # create axes object
    # (the variables point to new objects in each iteration)
    figure = plt.figure()
    axes = figure.subplots()
    # plot the data
    axes.plot(years, refugees[row], color="r")
    # add labels and title
    axes.set_xlabel("year")
    axes.set_ylabel("refugees")
    axes.set_title("Refugees in "+str(row))

# display the plot
plt.show()
```




Refugees in 1
refugees
year
140000
120000
100000
80000
60000
40000
20000
1990
1995
2000
2005
2010
2015
2020



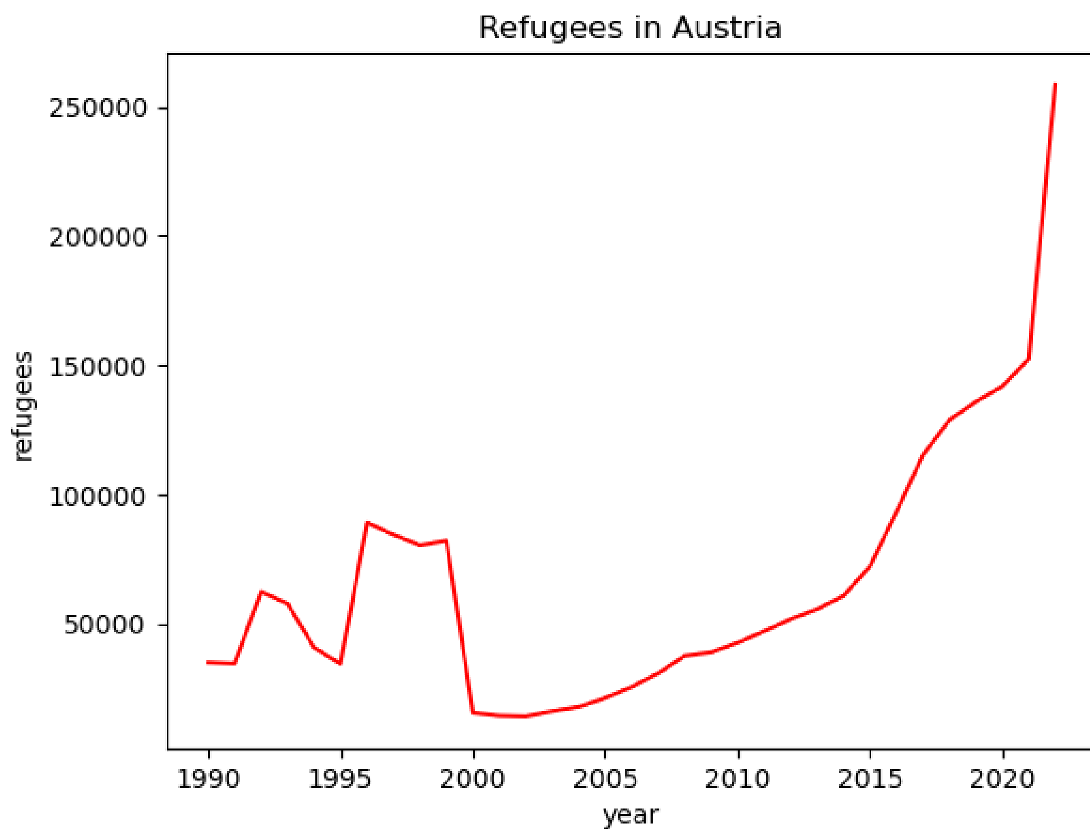
Of course we would like to use the proper country names. For the moment, we pick them from the file "manually"

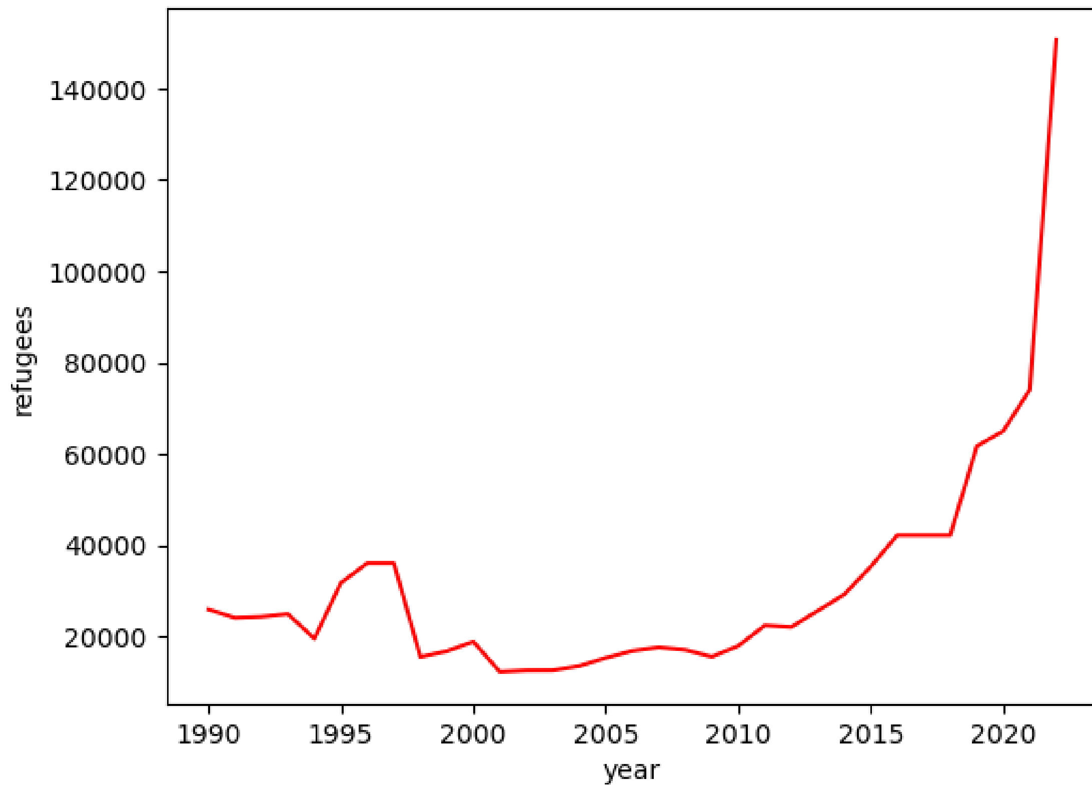
[93]:
```
!tail -n +2 data/countries-europe.csv | cut -d "," -f 2 | sed 's/^/"/' | sed 's/
↪$/",/' | fmt -w 1000
```

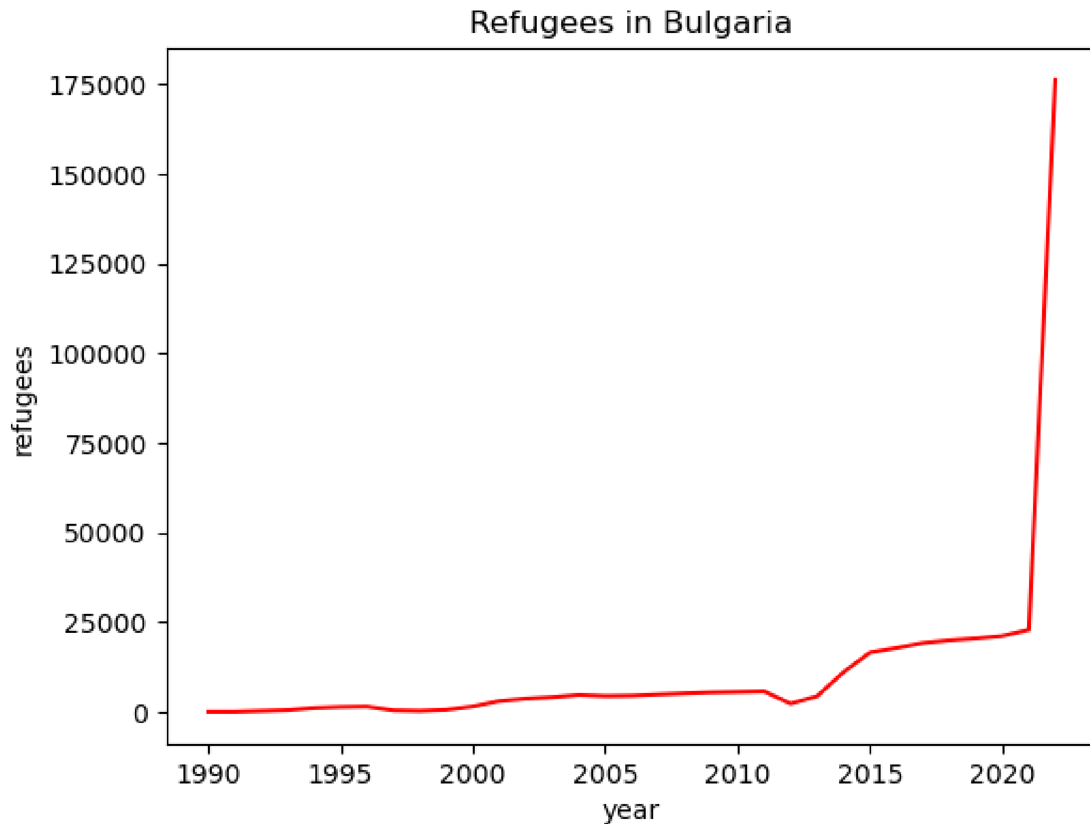
```
"Austria", "Belgium", "Bulgaria", "Croatia", "Cyprus", "Czechia", "Denmark",
"Estonia", "Finland", "France", "Germany", "Greece", "Hungary", "Ireland",
"Italy", "Latvia", "Lithuania", "Luxembourg", "Malta", "Netherlands", "Poland",
"Portugal", "Romania", "Slovak Republic", "Slovenia", "Spain", "Sweden", "United
Kingdom",
```

[94]:
```
import numpy as np
import matplotlib.pyplot as plt
# define data variables
years = range(1990,2023)
countries = ["Austria", "Belgium", "Bulgaria", "Croatia", "Cyprus", "Czechia",␣
↪"Denmark", "Estonia", "Finland", "France", "Germany", "Greece", "Hungary",␣
↪"Ireland", "Italy", "Latvia", "Lithuania", "Luxembourg", "Malta",␣
↪"Netherlands", "Poland", "Portugal", "Romania", "Slovak Republic",␣
↪"Slovenia", "Spain", "Sweden", "United Kingdom",]
```

```
refugees = np.loadtxt("data/refugees-europe_0.txt") # data from World Bank

for row in range(3):
    # create axes object
    figure = plt.figure()
    axes = figure.subplots()
    # plot the data
    axes.plot(years, refugees[row], color="r")
    # add labels and title
    axes.set_xlabel("year")
    axes.set_ylabel("refugees")
    axes.set_title("Refugees in "+countries[row])

# display the plot
plt.show()
```




Refugees in Belgium
refugees
140000
120000
100000
80000
60000
40000
20000
1990
1995
2000
2005
2010
2015
2020
year



This is a lot more efficient. We could still imporve a few things, but first we look at some other things.

4 More Control Flow Tools

We might want to limit our analysis on countries that have a large absolut number of refugees. To achieve this, we want to check that the average number of refugees is above 10000.

Of course we could manually create a list of these countries, but we will much rather use python.

4.1 If Statement

[95]:
```
for idx, mean in enumerate(refugees.mean(axis=1)):
    if mean > 100000:
        print(countries[idx], int(mean))
```

```
France 225212
Germany 900264
Sweden 154282
United Kingdom 171312
```

[96]:
```
for country, mean in zip(countries, refugees.mean(axis=1)):
    if mean > 100000:
        print(country, int(mean))
```

```
France 225212
Germany 900264
Sweden 154282
United Kingdom 171312
```

4.1.1 A Complex Example

[97]:
```
my_number = 0
```

[98]:
```
if my_number < 0:
    print("negative")
elif my_number == 0:
    print("zero")
elif 0 < my_number <= 3:
    print("small")
else:
    print("large")
```

```
zero
```

Note:

- `==` comparison; `=` assignement
- you can compare against a range

More:

- is / is not
- in / not in
- boolean operations
- float comparison

[99]:
```
if type(my_number) is int:
    print("It's an int!")
```

```
It's an int!
```

[100]:
```
if my_number in range(5):
    print("I found it!")
```

```
I found it!
```

[101]:
```
if type(my_number) is int and my_number in range(5):
    print("Two things at once.")
```

```
Two things at once.
```

[102]:
```
if np.sqrt(2)**2 == 2:
    print("Good")
else:
    print("Not so good! :-(")
```

```
Not so good! :-(
```

[103]:
```
np.sqrt(2)**2
```

[103]: 2.0000000000000004

4.2 Defining Functions

As mentioned at the beginning, once your program gets longer, you should splitt it into small, managable chunks.

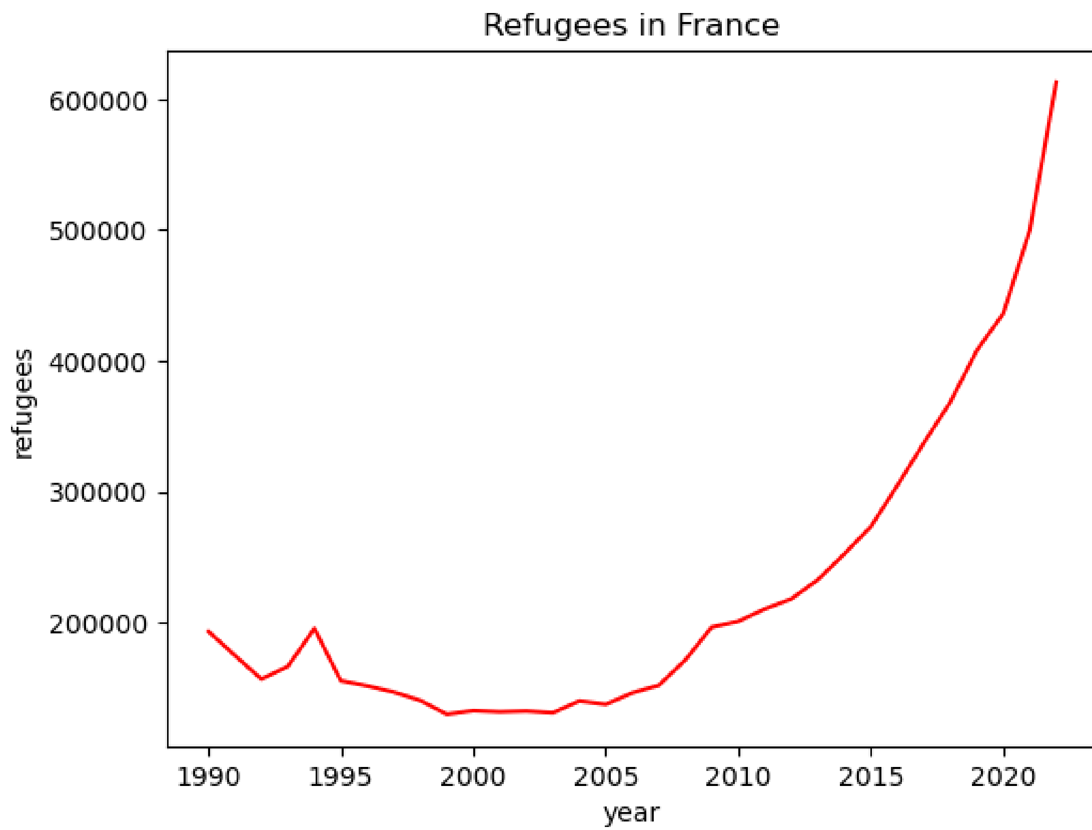
In our example, there is at least three tasks:

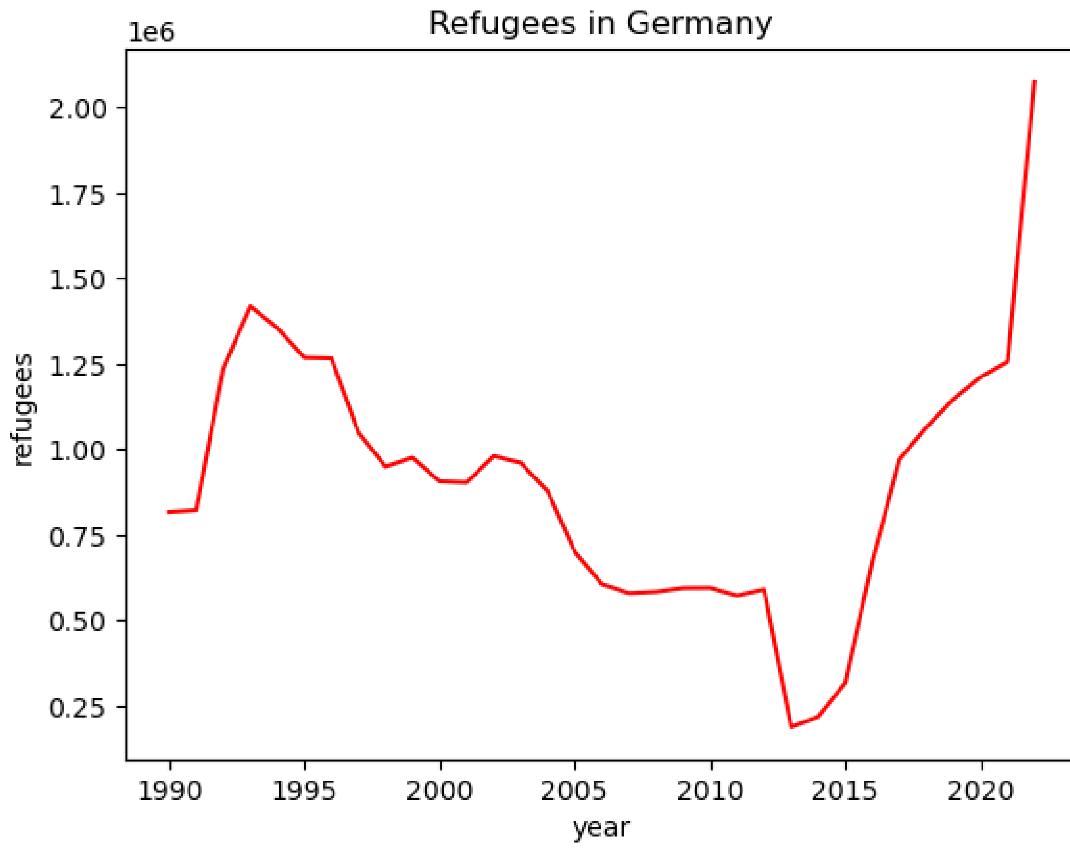
- defining the data variables
- plotting a single country
- managing several plots

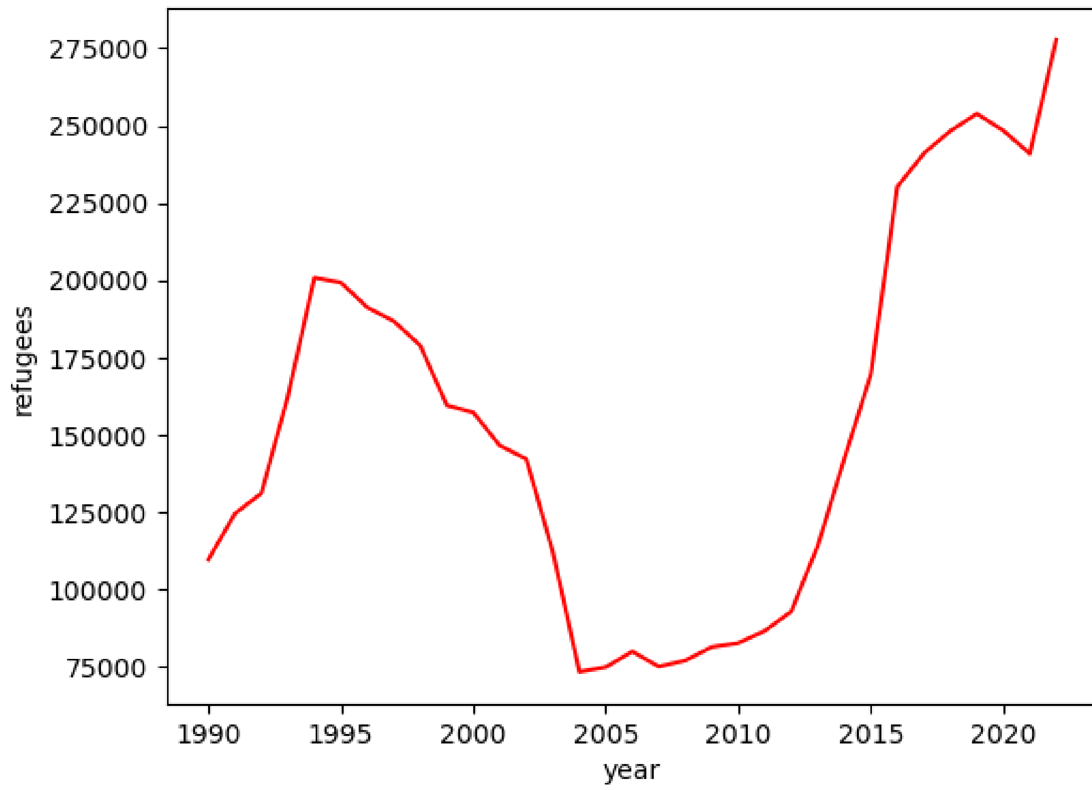
We followed this split when we developed our code. But we still put everything into one big code block. The next step is to define new functions for the spearate chunks. The obvious candidate for a function is the plotting for a single country.

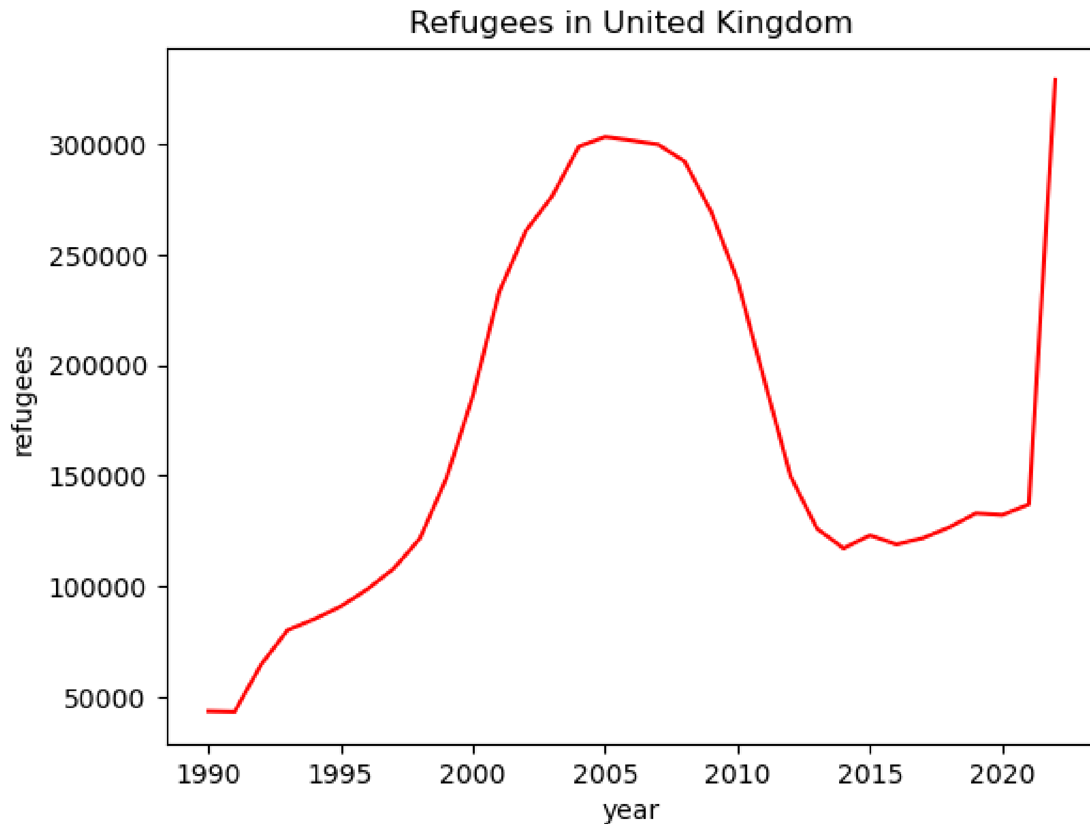
[104]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_refugees(years, data, country):
    # create axes object
    figure = plt.figure()
    axes = figure.subplots()
    # plot the data
    axes.plot(years, data, color="r")
    # add labels and title
    axes.set_xlabel("year")
    axes.set_ylabel("refugees")
    axes.set_title("Refugees in "+country)


# define data variables
years = range(1990,2023)
countries = ["Austria", "Belgium", "Bulgaria", "Croatia", "Cyprus", "Czechia",␣
 ↪"Denmark", "Estonia", "Finland", "France", "Germany", "Greece", "Hungary",␣
 ↪"Ireland", "Italy", "Latvia", "Lithuania", "Luxembourg", "Malta",␣
 ↪"Netherlands", "Poland", "Portugal", "Romania", "Slovak Republic",␣
 ↪"Slovenia", "Spain", "Sweden", "United Kingdom",]
refugees = np.loadtxt("data/refugees-europe_0.txt") # data from World Bank
```

```
means = refugees.mean(axis=1)
for row in range(len(countries)):
    if means[row] > 100000:
        plot_refugees(years, refugees[row], countries[row])

# display the plot
plt.show()
```




Refugees in Germany
1e6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
refugees
1990
1995
2000
2005
2010
2015
2020
year


Refugees in Sweden
refugees
year
275000
250000
225000
200000
175000
150000
125000
100000
75000
1990
1995
2000
2005
2010
2015
2020



A function has a fixed structure:

- The keyword **def** introduces a function definition.
- It must be followed by the function name (choose something helpful)
- and the list of formal parameters in round brackets.
- the definition line is then ended by a :
- The statements that form the body of the function start at the next line, and must be indented.

Functions define a scope for variables. This means:

- Variables we define inside a function (including the parameters) only exist inside the function.
- On the other hand, variables defined in the global scope will exist inside all functions.

[105]:
```
def outer_function(outer_arg):
    print("1:", global_value)
    print("2:", outer_arg)
    outer_value = "123"
    print("3:", outer_value)
    inner_function()

def inner_function():
```

```
    print("4:", global_value)
    #print("5:", outer_arg)
    #print("6:", outer_value)

global_value = "ABC"
outer_function("foo")
#print("7:", outer_value)
```

```
1: ABC
2: foo
3: 123
4: ABC
```

4.2.1 Our first examples do not return anything.

But like the predefined functions we have used so far, our own functions can return results too. The keyword for this is `return`.

[106]:
```
def with_interest(value, rate, years):
    return value*(1+rate/100)**years

def compound_interest(value, rate, years):
    return with_interest(value, rate, years) - value

def info(value, years):
    rate = 0.5
    print("Compound interest:", compound_interest(value, rate, years))

balance = 1000
res = info(balance, 5)
#print("info returns:", res)
```

```
Compound interest: 25.25125312812429
```

Make sure you understand the difference between print and return

Obviously this code is not very elegant. Once you understand the difference between `print` and `return`, you could get rid of the info function completely.

5 User Input

Of course we can hard-code the balance in our last exampe (put the number into the code). But it would be a lot nicer, if our script would ask the user for the balance to use. The easiest way to achieve this is, to ask the user during execution. To do this, Python offers the `input` function.

[107]:
```
name = input("What is your name? ")
print("Hello", name)
```

```
What is your name? Nicola
Hello Nicola
```

The return value of the `input` function is always a string, no matter what you enter:

[108]:
```
input_ = input("Your input? ")
type(input_)
```

```
Your input? 123
```

[108]:
```
str
```

So if you want to read a number, you will need to convert the output of `input`. Luckily this is easy

[109]:
```
number_string = input("Enter a Number? ")
my_float = float(number_string)
print(my_float+1)
```

```
Enter a Number? 123
124.0
```

With this we can adapt the compound interest script

[110]:
```
def with_interest(value, rate, years):
    return value*(1+rate/100)**years

def compound_interest(value, rate, years):
    return with_interest(value, rate, years) - value

balance = float(input("Enter the balance: "))
time = int(input("Enter the number of years to accumulate interest for: "))
interest_rate = float(input("Enter the interest rate (%): "))
res = compound_interest(balance, interest_rate, time)
print("Compound interest:", res)
```

```
Enter the balance: 1000
Enter the number of years to accumulate interest for: 10
Enter the interest rate (%): 2
Compound interest: 218.99441999475744
```

6 Hands On 3

6.0.1 Questions

- What is the basic structure of a for-loop?
- What type of objects can you use a for loop on?
- Explain the functions `enumerate` and `zip`.
- How do you use `if`, `elif`, `else`?
- How do you define a function?
- What variables are availabel in the different functions of the last example.
- What is the difference between `return` and `print`?
- What is intendation good for?

6.0.2 Tasks

- Repeat the things we did, type in the commands yourself and experiment with them.
- Write a loop that calculates `2 ** 8` with simple multiplication.
- Look at the help for `range` and create a loop that prints a countdown.
- Adapt the compound interest code:
  - let it use a sensible default value for `balance` and `time` if these values are not specified.
  - let it calculate the result for a fixed list of interest rates (e.g.: 0.5, 1., 2., 4., 8.) instead of reading a single value from the terminal.
  - calculate the results for each interest up to 15 years and plot the result.
- Define a function that calculates some basic statistics (min, max, mean, standard deviation) for the passed data and *returns* a list with them.
- Use the function from the previous task to print these statistics for all countries where the mean is above 10000.
- Adapt the refugees code such that the user can choose the time range to plot.