IP 'Python in Geosciences': Climate Emulator in CDICE

Doris Folini, Mathias Hauser, Alessio Ciullo

February 24, 2024

Abstract

The IP on Python in Geosciences (IPP) invites you to gain hands on experience on some aspects of python. In parallel with simple examples specific to the IPP, you will encounter examples taken from a concrete, real world application context. This mix is deliberate, the practicum aims at teaching some basics, but also wants to expose you to real world problems. For the same reason, you should form teams of two - in reality, coding is often a joint effort and / or others should be able to use what you coded.

You get eight tasks to work on, which address different python topics and go from rather simple to more and more difficult. All tasks can be done in teams, one tasks is explicitly meant as team work. Some tasks (or parts of tasks) are marked with "For hand-in" and you are expected to submit us your python scripts for these. Associated parts of tasks are explicitly marked by 'For hand-in'. Also, at the end of the IPP you must present during about 15 minutes what you did (explain some of your code, associated highlights or challenges, what did you pick from the non-compelling tasks, how did you solve that, what aspects of python you think are really cool / useful or, by contrast, missing). **You are not expected to solve all tasks / all parts of each task - or to understand everything.**

Contents

1 Science Context

The reports on climate and climate change published by the Intergovernmental Panel on Climate Change (IPCC) form an essential pillar of the associated political, economic, and societal debate. Within each IPCC report, Working Group I (WG I) focuses on 'The Physical Science Basis', on what natural sciences can tell us about climate and climate change. This working group relies heavily on observational data, as well as on simulation data from Global Climate Models (GCMs, featuring an atmosphere and and ocean) and Earth System Models (ESMs, featuring also an interactive carbon cycle). Such models cover in great detail a wide variety of physical, biological, and chemical processes. As such, they may be seen as some sort of a gold standard in climate science. The downside is that these models are very expensive to run on a computer. On a modern high performance computer it typically takes several hours to produce just one year of simulated climate. The simulations entering the IPCC reports typically take several months.

This is too expensive for other applications and communities that are likewise relevant in the climate change debate. A concrete example are studies that address economic aspects of climate change and associated feedbacks. For example, what are the economic costs of climate change or how should an effective carbon tax be designed? Studies of this sort require a detailed description (modeling) of the economy and, consequently, must compromise on the climate model part. In practice, this means that instead of an expensive GCM or ESM (see above), a much simpler Climate Emulator (CE) is used. As the naming suggests, a CE is meant to emulate (mimic) some of the (complex) behavior (results) of GCMs. A prominent example is the change in global mean temperature in response to carbon emissions. An ESM simulates this response in great physical and biogeochemical detail. The CE consists of only a few equations (see below) and is calibrated to mimic the response of the ESM to carbon emissions: what fraction of carbon emissions remains in the atmosphere (i.e., is not taken up by the ocean or the land biosphere), thereby changing the atmospheric CO_2, thereby causing a greenhouse gas forcing, thereby changing (increasing) the global mean temperature. Such a simple CE can then be used to study the interplay between economy (emitting carbon) and climate (where carbon emissions cause temperature to increase, resulting in economic losses / damages).

The Nobel Prize in Economic Sciences in 2018 was awarded for pioneering work in this field to William Nordhaus (shared with Paul Roemer for his work on endogenous growth theory). The CE of the associated model, the model being named DICE (Dynamic Integrated Climate-Economy model, see Figure 1 for a sketch), forms the scientific context of this specific project within the IPP. Put simply, the project gives an impression of what a simple CE like in DICE can do, and what it cannot do. A detailed description of the model can be found in Folini et al. [2024]. The python code used was written in this context. The model comprises two parts: a carbon cycle, which 'translates' carbon emissions into atmospheric CO_2 concentrations, and a climate part, which 'translates' changes in atmospheric CO_2 concentrations into changes in global mean temperature. The question then arises what such a cheap but simple CE - simple climate model - can do and what it can not do.

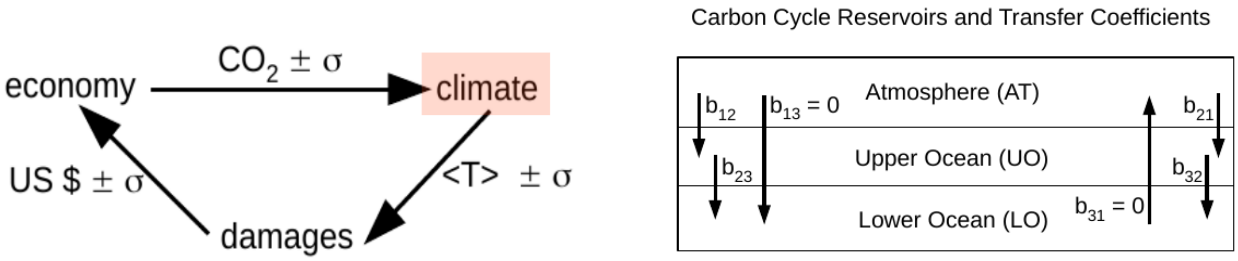


Figure 1: **Left:** Schematic view of coupled economy-climate model. The economy (top left) produces CO_2 emissions, which serve as input to a CE (simple climate model, top right). The CE 'translates' the CO_2 emissions into an atmospheric CO_2 concentration and an associated change in global mean temperature, $< T >$. This temperature change is used to estimate climate related economic damages (bottom) that feed back as a cost factor to the economy (top left). All quantities may be accompanied by some uncertainty $\pm\sigma$. The CE part of this feedback loop provides the scientific context of this specific project within the IPP. **Right:** Illustration of the three reservoirs of the carbon cycle and associated transfer coefficients.

Formally, the carbon cycle consists of three reservoirs (atmosphere, upper ocean, lower ocean), whose carbon masses (in GtC, Giga tons Carbon) $\mathbf{M}_t = (M_t^{\mathsf{AT}}, M_t^{\mathsf{UO}}, M_t^{\mathsf{LO}})$ are updated from time t to $t+1$ (time step Δt) via

$$\mathbf{M}_{t+1} = (\mathbf{I} + \Delta t \cdot \mathbf{B}) \cdot \mathbf{M}_t + \Delta t \cdot \mathbf{E}_t, \tag{1}$$

with $\mathbf{E}_t$ emissions[1] at time t, $\mathbf{I}$ the identity matrix, and $\mathbf{B}$ the transfer matrix among the reservoirs,

$$B = \begin{pmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \\ b_{13} & b_{23} & b_{33} \end{pmatrix}. \tag{2}$$

The climate part consists of two coupled ordinary differential equations for the temperatures of atmosphere and ocean,

$$T_{t+1}^{\mathsf{AT}} = T_t^{\mathsf{AT}} + \Delta t \cdot c_1 \left(F_t - \lambda T_t^{\mathsf{AT}} - c_3 \left(T_t^{\mathsf{AT}} - T_t^{\mathsf{OC}} \right) \right), \tag{3}$$

$$T_{t+1}^{\mathsf{OC}} = T_t^{\mathsf{OC}} + \Delta t \cdot c_4 \left(T_t^{\mathsf{AT}} - T_t^{\mathsf{OC}} \right). \tag{4}$$

The c_i are (physical) constants, the radiative feebdack parameter λ is the ratio of the forcing $F_{2x\mathrm{CO2}}$ from a doubling of CO_2 to the (equilibrium) temperature change $T_{2x\mathrm{CO2}}$ under such a doubling, F_t is any non-CO_2 forcings, e.g. from methane or aerosols.

[1]For example, emissions from fossil fuels in 2020 are estimated at 34.8 $GtCO_2$ or 9.5 GtC, thus $\mathbf{E}_t = [9.5, 0, 0]$; see https://www.co2.earth/global-co2-emissions or also Le Quéré et al. [2018]

2 Python Code

The python code used here was originally written to systematically test the CE described above and produce publication ready figures [as used in Folini et al., 2024]. The choice to use this code directly (after some cleaning and removal of non-essential contents) for this IPP was deliberate: it is through really existing, practical applications that you learn about practical work. The code you get is primarily 'science code', not 'teaching code'. As such, it also violates here and there 'good coding practice rules'[2]. The idea is that you learn to work within a given, pre-existing framework, to use and adapt something existing to your needs. This is a situation you are likely to encounter over and over again in your career. It is not necessary that you look at (let alone understand) each and every line of code. Although you are certainly welcome to do so during this IPP if you wish. The code of the CE is organized into four *.py files, *python scripts* and *python modules*:

1) Figs4Paper.py - definitions of specific, reproducable figures for publications.
In python terms, a script, a collection of statements (if-statements, function calls)

2) TestDefs.py - definitions of specific tests and associated plotting functions.
In python terms, a module, a collection of function (method) definitions

3) ClimDICE.py - climate emulator as such, carbon cycle and temperature equations.
In python terms, a module, the definition of a class (containing methods)

4) PatternScaling.py - utilities to scale global mean temperature change to 2D map.
In python terms, a module, a collection of function (method) definitions

In addition, you are provided files specific to this IPP: globals.py, Task*.py, Chiens.py (details are given in the individual tasks). The codes will allow you, in particular, to familiarize yourself with a few aspects of python, including: **basic inquiries** (shape, type); **basic flow control** (if-else, for-loops); **basic modules** (numpy, re, os); **matplotlib** (plotting lines); **cartopy** (plotting geographical maps); **pandas** (bundling data into data frames instead of simple arrays); **function definition** (bundling of statements); **class definition** (bundling of functions).

A word on python scripts and modules (*.py files). Python scripts are just a way to collect (wrap) individual python commands (statements) in a (more or less) orderly way. Instead of typing each command separately, you can just 'run' or 'execute' the python script (the *.py file). This not only saves you work (typing). It also helps you reproduce what you did (you have all the commands you used in the *py file) and then re-use what you did in a different context. Also, it allows you to easily share what you did (the *.py files) with others - be it that they want to reproduce what you did or they want to profit from the time you already invested for their own work, they want to re-use your work and adapt it to their purpose.

Instead of 'only' collecting the series of commands you would type at a command prompt, you can bundle series of commands you use over and over again in a function - and then use that function in place of the series of commands. These functions you may store away again in a *.py file, in a python module that you can import when you want to use your functions.

All these advantages of python scripts and modules (reproducability, reusability, sharing with others) depend heavily on the 'quality' of the scripts: how well they are organized, how clean they are written, how well they are commented. Python helps you in this endeavor by providing a wealth of useful

[2]For example, you will find multiple statements on one line, separated by a ;, which is not considered good coding practice and actually should be avoided. Also, there may be unused variables, indenting does not always follow 'good practice' etc. For the curious and ambitious (thanks to Mathias Hauser for the hint): you can use 'flake8' to learn more about the deficits of the code. No help here from our side.

'language elements'. Data can be organized, for example, in list, (multidimensional) arrays, tuples, dictionaries, data frames (pandas, xarray). Methods to work with these data can be organized in functions or classes.

The concrete code of the CE you use in this practicum illustrates these points. You will also see that the code is far from perfect. A main reason here is that the code 'grew organically' with the science task at hand, a typical situation in real day life. Given man power (time) as the limiting factor, one has to compromise on where to invest how much time: further developing the capabilities of the code, keeping it readable, see that its parts can be re-used, have it sufficiently well commented.

3 Task by Task

This project within the IPP is organized in seven tasks, described one by one below. Each task should take you roughly one half day of the practicum, depending on how much you know and how deep you want to dive in each task. It is recommended to start with task 1, then go to task 2 etc. However, if you already feel somewhat confident with python, you may change the suggested series of tasks or skip one or the other. Individual tasks are reasonably 'stand alone' - if you already know some python. How far you get with all the tasks also depends on your previous knowledge - there may be too many tasks for you or too few. The goal is that you learn 'real life' python with the help of these tasks and present in the end (Leistungskontrolle!) some highlights of what you did, what you learned - or where you failed.

To get started: a) use the conda environment 'ipp_analysis' (see introdcution) and b) download codes and data (zip-files) from the course web-page to your laptop. Put the zip-files with the code and data in a folder. Go to that folder and unzip what you downloaded. You should get the following folders: 'Code', 'DataFromCMIP', 'EmiAndConcData', 'pattern_library' and 'pattern_library_cg'. The folder 'Code' contains all the *py files you will need. 'DataFromCMIP' contains CMIP5 data used for bench marking the CE. 'EmiAndConcData' contains data on CO_2 emissions and concentrations used in CMIP5. The folder 'pattern_library' contains spatial patterns to 'translate' a change in globale mean temperature into a 2D temperature change pattern (featuring e.g. that poles warming more than the equator, or land warms more than ocean). The folder 'pattern_library_cg' contains the same patterns, but remapped onto a common, coarse grid.

3.1 Task 1: dir(), type(), help(), matplotlib

This task has three goals: a) give an impression of the IPP's science context (just run an existing python script and look at the figures you get); b) introduce some useful built-in python commands (*dir(), type(), help()*); c) produce a figure yourself (use what you learned in the introduction to data from the science context).

a) Go to where you put the *.py files. Launch ipython by typing at your command prompt *ipython*

b) Science context of the IP. Type *%run Figs4Paper.py*[3]. You should get three plots on your screen: CO_2 concentration and temperature change as function of time (year), as well as a temperature change map for one specific year (2080). The data shown come from the CE described in Section 1. Later tasks will dive more into the CE. For now, just look a bit at the plots and think for

[3]Python scripts can be run in different ways, including: from within jupyterlab (see introduction), from within an ipython command prompt using '%run MyScript.py', from within a python command prompt using 'exec(open("./MyScrip.py").read())', at the command prompt of your computer as such using 'python MyScript.py'. For the last case, the first line in MyScript.py must read something like *#!/usr/bin/env python3*. For more details see e.g. here, https://realpython.com/run-python-scripts/

yourself what the plots tell you. You can close the plots on your screen again using 'plt.close("all")' from the matplotlib module (first import the module by typing 'import matplotlib.pyplot as plt').

c) Get an idea of what is around using some built-in python functions, *dir(), type(), help()*. Type *dir()*. What do you see? Learn more about specific variables (e.g. sisi or T_AT). Type *dir(sisi)* or *type(sisi)* or *help(sisi)*. What do you see? How about other variables?

d) Examine more closely the variables 'Y_AT' and 'T_AT' (actually 'globals.Y_AT' and 'globals.T_AT'; outputs from the CE, year and global annual mean temperature change of the atmosphere, relative to year 1850). What variable type are they? What is their dimension (length, size, shape)? What is the array index of year 1980? [Hint: do it by hand, using 'try and error', or use numpy.where()] What is the atmospheric temperature change T_AT in 1980? What is the temperature change averaged over 1960 to 1989?

e) Use matplotlib (see introductory material) to plot T_AT (y-axis) as function of Y_AT (x-axis). [Hint: plt.figure() and plt.plot() ; do not forget to import (parts of) matplotlib] Repeat the plot, but change it [line color / style / width; annotate the x- and y-axis; add figure title; add a grid; set axis range by hand; change font sizes; save figure to png-file]

f) For hand-in, Task1.py: Put what you did under point e) in a python script file 'Task1.py' that you can run after having run Figs4Paper.py [Hint: Use the Task1Skeleton.py file as a starting point. Rename it to Task1.py. Make sure you have globals.py in the same directory / folder. To run, use again the syntax *%run Figs4Paper.py and %run Task1.py*.]

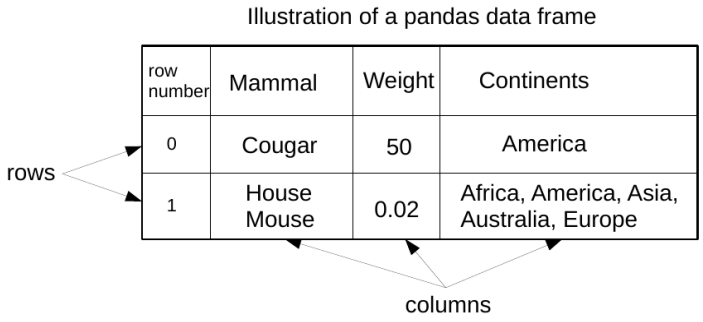
3.2 Task 2: pandas data frames

The CE uses pandas data frames[4] to bundle various information for a specific test case. Pandas data frames are useful as they allow you to bundle all sorts of information. In the case of the CE this comprises, for example, basic information on the test as such (e.g. what test problem?), concrete choices of parameter values, benchmark data (what is the expected outcome of the test?), but also results from the simulations that are being tested against these benchmark data. In terms of python, pandas data frames allow you to bundle strings, floats, (numpy) arrays, lists, but also more complex structures like dictionaries or, as you will learn, instances of classes (you already encountered an instance of a class: *sisi* in task 1).

Basically, you can think of a pandas data frame as a two dimensional table with rows and columns. Each column designates some property to be listed in the data frame. A three column example could be the name of a mammal species, its average weight in kg, and on which continent(s) it can be found. Each row is an entry to the data frame. Sticking with the example, a first row could be 'cougar', 50, 'America'. A second row may be 'house mouse', 0.02, 'Africa, America, Asia, Australia, Europe'. As you can see, a single entry may combine a single name (string with or without blanks), a number (an integer or a float), and a list of names (the different continents). The data frame can be accessed in various ways: get all data on the house mouse or get the range of weights of all the beasts listed.

The present task aims at familiarizing you with pandas data frames. The task consists of two sub-tasks: after a short introduction to the basics of pandas data frames (sub-task 1), you should go to the TestDefs.py from the CE and add additional simulations (e.g. different choice of simulation parameters) to the pre-defined tests (e.g. CMIP5 RCP85; Coupled Model Intercomparison Project Phase 5; Representative Concentration Pathway for a forcing of 8.5 W/m^2 in 2100).

[4] A relatively simple intro to pandas data frames may be found here:
https://www.tutorialspoint.com/python_pandas/python_pandas_dataframe.htm




```
data = [['Alex',10],['Bob',12],['Clarke',13]]            #data going to data frame [one python list per pandas data frame row]
df = pd.DataFrame(data,columns=['Name','Age'])  #combine data and column headers into data frame
print(df)

data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print(df)

keys = ['Model', 'ECS', 'linestyle', 'linecolor']                    #python list of column names of your data frame
data = []                                                                          #empty python list; each list entry will be a row in the data frame
data.append( ['HadGEM2-ES', 4.55, 'solid',  [0.0, 1.0, 1.0]] ) #add list entry for CMIP5 model HadGEM2-ES
data.append( ['GISS-E2-R',  2.15, 'dashed', [1.0, 0.7, 0.0]] ) #add list entry for CMIP5 model GISS-E2-R
data.append( ['MPI-ESM-LR', 3.65, 'solid',  [1.0, 0.0, 1.0]] )  #add list entry for CMIP5 model MPI-ESM-LR
df=pd.DataFrame.from_records(data, columns=keys);      #combine data and column headers into data frame
print(df)
```

Figure 2: **Top:** Sketch of pandas data frame. **Bottom:** Three python code examples of (simple) pandas data frames. See Task 2.

a) Go to where you put the *.py files. Launch ipython by typing at your command prompt *ipython*

b) Get an impression of pandas data frames in the CE. Type *%run Figs4Paper.py*. This will return you a pandas data frame, named *df2*. Examine it. Start with simply typing *df2*. What do you see? Type *df2.c4* or *df2.label* or *df2.c4[1]* - what do you see? Use what you learned in task 1 (built-in python functions *dir(), type(), len(), help()*) to get a better impression of df2 and of what it contains. To see more details, you may also go to the Figs4.Paper.py file and search for *df2=td.BenchMarkTDEmiOrConc(CMIP='RCP26_EMI')*.

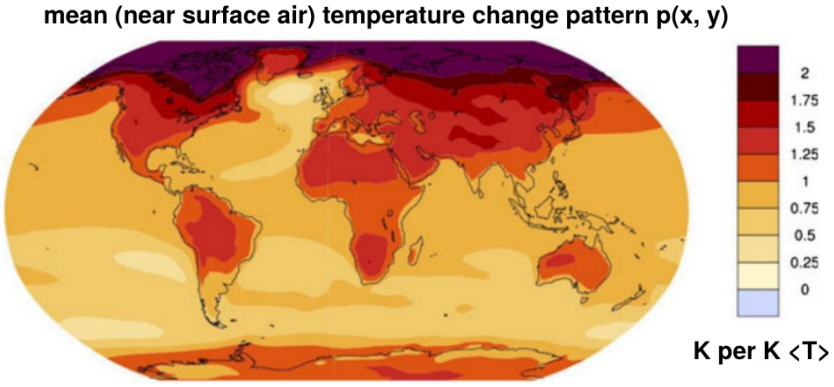
c) For hand-in, Task2.py: Build your own pandas data frame. Use the file 'Task2Skeleton.py' as a starting point. Copy it to 'Task2.py' and add the example data frame code given in Figure 2. Also add a data frame for the data in the top part of Figure 2 (mammals), add some more columns to it (e.g. average age or heart beat rate). Examine the data frames you just created. For example, type *df* or *df['Model']* or *df['Model'][0]* or *df['Model'][1:2]* or *df['linecolor']* or *df['linecolor'][:]* or *df['linecolor'][:]0]*. What do you see?

d) Change an existing pandas data frame in the CE. Open the file TestDefs.py and search for *if (prb=="TD_emiCO2"):*. There you see pandas data frame entries for the test simulations you saw before, in the figures produced by *%run Figs4Paper.py*. In these figures (maybe re-run Figs4Paper.py), can you say which line belongs to which entry in the pandas data frame? Duplicate one of the existing entries (e.g. the one with label 'CDICE-HadGEM2-ES'), then modify the duplicated entry slightly (change the entry for 't2xco2' from 4.55 to 6.66; change the linestyle entry 'ls' from 'solid' to 'dashed'). Check what this does by executing again Figs4Paper.py, i.e. type again *%run Figs4Paper.py*. What do you see?

```
import matplotlib.pyplot as plt              #basic plotting stuff in python, e.g. create a figure window via plt.figure()
import cartopy.crs as ccrs                   #cartopy coordinate reference systems (crs) [PlateCarree [equidistant lon/lat], Mercator, Mollweide...]
import cartopy.feature as cfeature           #cartopy features, e.g. borders, rivers, lakes, coast lines
from cartopy.util import add_cyclic_point #cartopy utilities, e.g. add a cyclic point to identify 360 degrees and 0 degrees longitude

print("cartopy: draw coastlines using PlateCarree projection [equidistant longitude / latitude]")
plt.figure(figsize=(10,6))                    #open a figure window using matplotlib plt.figure()
ax = plt.axes(projection=ccrs.PlateCarree())  #use the cooridnate reference system (crs) from cartopy to define a map projection
ax.coastlines()                               #using this coordinate system (coordinate axis), plot coastlines
plt.show(block=False)                         #show the plot on the screen

print("cartopy: draw coastlines using Mollweide projection, add lakes and rivers")
plt.figure(figsize=(10,6))                    #open a figure window using matplotlib plt.figure()
ax = plt.axes(projection=ccrs.Mollweide())    #use the cooridnate reference system (crs) from cartopy to define a map projection
ax.coastlines()                               #using this coordinate system (coordinate axis), plot coastlines
ax.add_feature(cfeature.LAKES)                #add lakes
ax.add_feature(cfeature.RIVERS)               #and rivers to your map
plt.show(block=False)                         #show the plot on the screen
```



[Tebaldi & Arblaster (2014), Fig. 3a]

Figure 3: **Top:** Cartopy map projection examples. See Task 3. **Bottom:** Pattern scaling. Regional temperature change per degree global mean temperature change <T>. If the global mean temperature increases by one degree, poles warm more, the equator less. Land warms more than oceans.

3.3 Task 3: cartopy

It is often convenient, even necessary, to show data on a geographical map. One way to do so in python is to use the module cartopy. The goal of this task is to introduce you to cartopy. The task again consists of essentially two parts. In a first part, you look only at map projections *without any data*. This includes drawing coast lines, rivers, and other geographical data that comes with cartopy. Get an idea of the many different ways the 3D Earth can be projected onto a 2D map.

In the second part, you will add data to the 2D map projection. More specifically, you will add 2D temperature data from the CE, described in more detail below. Note that you now have to take care about two coordinate reference systems: the one describing your data (the temperature data will come on an equidistant latitude / longitude grid) and the coordinate system that comes with the map projection you choose (e.g. Mercator is a well known map projection). You inform cartopy about both coordinate systems: what map projection you would like to see on your screen, and what coordinate system the data uses that you wish to plot. With this information given, cartopy can then transform your data to the map projection you wish to see. Note that typically you cannot change the coordinate reference system of your data. The data and the its coordinate system belong to each other, they cannot be separated. But you are typically free to pick whatever map projection you would like to see on the screen.

The concrete 2D temperature change data from the CE is obtained via pattern scaling. The CE

computes the change in global mean temperature for changing atmospheric CO2 concentrations, for example following the RCP85 from CMIP5. But global warming is not spatially homogeneous: the poles warm more strongly than the equator. This behavior can be approximately captured by pattern scaling, by providing a 2D map showing the local warming per degree warming in the global mean. More details can be found in the file PatternScaling.py

a) Go to where you put the *.py files. Launch ipython by typing at your command prompt *ipython*

b) Science context. Get an impression of maps (done using cartopy) in the CE. Edit the file 'Figs4.Paper.py': set *F_RCP26E = 'no'* and *PS_RCP85E = 'yes'*. Run the script by typing *%run Figs4Paper.py*. You should see a map showing the temperature change under RCP85 in the year 2080, with respect to the base year 1850. Take a moment to look at the map. What do you see?

c) For hand-in, Task3.py: Draw your own maps. First only the map as such (coastlines etc.), then add data (temperature change as you saw it under point b). Use the file 'Task3Skeleton.py' as a starting point. Copy it it to 'Task3.py'. Add example code as given in Figure 3. Run your code. Check a bit what is around. Use *dir()* to examine what coordinate systems the Coordinate Reference Systems (crs) module of cartopy offers. Use *type()* to convince yourself that crs is a module. Draw (add to 'Task3.py') global maps with coastlines in different projections. Add features (e.g. rivers or borders) to these maps using the feature module from cartopy (*cartopy.feature*). Use again *dir()* to examine what features are available from cartopy. Likewise, use *dir()* and *help()* to find out what possibilities you have with your coordinate axis (*ax=plt.axes(projection=ccrs.PlateCarree()) ; dir(ax) ; help(ax)*)[5]. Add gridlines for longitude and latitude.

d) For hand-in, Task3.py: Now add temperature data to the maps. Running 'Figs4Paper.py' (see point b) should have got you arrays PS_lons, PS_lats, and PS_T2D_scal (and others; PS for Pattern Scaling). Check their shape, size, type. Convince yourself that the data come in an equidistant longitude / latitude (i.e. PlateCarree) coordinate reference system.
Now you have two coordinate reference systems: one from the data (you cannot change it; here it is PlateCarree) and one from the map projection you choose for plotting (this you can choose; take e.g. Mollweide). Extend the 'Task3.py' script by yet another plot that combines data and map. Use what you learned under point c and in the introduction. Use a Mollweide projection for the map and plot the temperature data onto the map [Hint: *ax.contourf(lons, lats, T2D_scal, levels=10, cmap='Reds',transform=ccrs.PlateCarree())*] Finally, try to add yet more things to your plot: location of Zürich; hatching; contour lines; change the projcetion / number of contours / the color map; add a color bar and annotate it. Ask google for help or check out the cartopy page. Or look at the function *PlotTempPat* in the file 'PatternScaling.py'.

3.4 Task 4: netcdf files

In geosciences, *netcdf*[6] is a wide spread data format (you are probably familiar with other data formats, like **.csv, *.txt, *.pdf*). An advantage of this file format is that it allows quick and easy access to its data, even when the file is very big. This is achieved by 'clever organization' of the data within the file. Basically, you can think of the contents of a netcdf file being organized not unlike the files and folders

[5] For more information on projections, check out https://scitools.org.uk/cartopy/docs/latest/reference/projections.html

[6] It is worth noting that the *netcdf* data format is closely related to *hdf*, the hierarchical data format. In fact, modern versions of netcdf are under the hood actually hdf5 files, as mentioned e.g. here
https://xarray.pydata.org/en/stable/user-guide/io.html

```
import glob          #unix style pathname extension; pathlib may be an alternative
import re            #regular expressions, to extract model name from path to pattern file

#netcdf file you want to open and read by hand
fname = "../pattern_library/PATTERN_tas_ANN_MPI-ESM-LR_rcp85.nc"
ds = xr.open_dataset(fname)  #open and read the netcdf file
xr.Dataset.close(ds)         #close the file again
pava = ds.pattern.values     #get the values of 'pattern' stored in ds

#list files matching pattern to then open in a loop
tmp  = sorted(glob.glob("../pattern_library/PATTERN_tas_ANN_*_rcp85.nc"))
ltmp = len(tmp)                  #how many files in list?
for i in range(ltmp):            #loop over all the files in the list
   ds   = xr.open_dataset(tmp[i]) #open the file
   nlon = len(ds.lon)             #get length of the array containing longitudes
   ...       #get also the latitudes and close the file again
   print("File %60s has %4i longitudes and %4i latitudes" % (tmp[i], nlon, nlat))

#storing part of what you read in a pandas data frame
for i in range(ltmp):            #loop over all the files in the list
   ...
   patt = ds.pattern.values      #get the pattern
   pts  = re.split('_',tmp[i])   #extract model name from tmp[i]; split tmp[i] at each occurance of '_'
   lpts = len(pts)               #count how many parts (lpts) we got in this way
   mnam = pts[lpts-2]            #model name is the second last part, i.e., array index lpts-2
   data.append( [mnam, nlon, nlat, patt, clim] ) #add info to data part of pandas data frame
   ...
df=pd.DataFrame.from_records(data, columns=keys)#combine data and column headers into data frame
#print(df.Model);  pp = df.Pattern[10]; pp.shape; type(pp); print(pp[20][30])
```

Figure 4: Code snippets for use with task 4, devoted to netcdf files.

on your computer[7]. The goal of this task is to illustrate this point by letting you play a bit with netcdf files. The task also introduces two other useful aspects of python: getting a listing of files on disk and 'regular expressions'. The latter can be used, for example, to find patterns in a given string.

a) Go to where you put the *.py files. Launch ipython by typing at your command prompt *ipython*

b) **For hand-in, Task4.py:** Get hands on experience with netcdf files. Use the file 'Task4Skeleton.py' as a starting point. Copy it it to 'Task4.py', then complement the three examples sketched with the help of the code snippets in Figure 4. In the first example, get info on *ds* by just printing it. What data type is *pava*? Print an individual value of *pava*, e.g. for the indexpair 20,30. In the second example, have the number of longitudes and latitudes printed. In the third example, use pandas data frame columns *'Model', 'Nlon', 'Nlat', 'Pattern', 'Climatology'* and fill the data accordingly. [Hint: for the model name, check out Figure 4; for the climatological temperature pattern, go back to the first example in this task, check out the contents of one data file by hand.]

3.5 Task 5: functions

Python allows you to define functions (or methods). This is useful to cleanly pack away 'stuff' that is re-occuring, that you want to re-use. The present task first provides some introduction to functions via (very) simple examples (see Figure 5). Subsequently, you are encouraged to use what you learned in tasks 1 to 3 (some basics of python, matplotlib, pandas, cartopy) to a) play with the CE and b) add some new plotting functions to it.

a) Go to where you put the *.py files. Launch ipython by typing at your command prompt *ipython*

b) Start with a simple python script, not yet a function. Type *print('hello')* and hit enter. What do you see? Pack what you just typed into a file 'hello1.py'. This is a python script, a collection of commands you could also just type on the command prompt (see Figure 5, top left, hello1.py). Run the script.

[7]For hdf5, this is nicely explained e.g. here
https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5

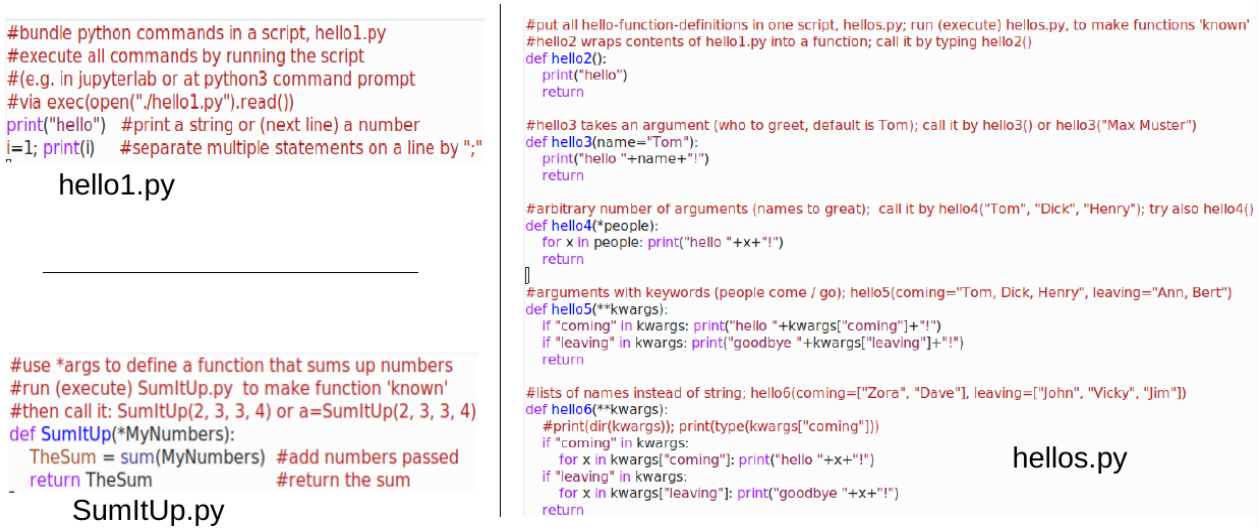


Figure 5: Example function definitions (hellos.py and SumItUp.py; hello1.py is not a function but just an ordinary python script). See task 5 for details.

c) For hand-in, hellos.py: Now, define a set of *functions that say 'hello'* and use them. Look at Figure 5, right. Pack the function definition for hello2 etc. into one file, 'hellos.py'. This collection of functions is a python module. One way to make 'visible' the functions you just defined is to run 'hellos.py'. Much like you run an ordinary python script (see point a above). Once you have done this, you can use your functions by just typing *hello2()*. Try it. Check out all the hello functions. For the **kwargs example, you may find more interesting examples on the internet that give a better idea of how useful **kwargs can be.

d) The standard (and more elegant!) way to make 'visible' the functions defined in 'hellos.py' is to use *import*. Exit ipython and re-launch ipython. Type *import hellos as hs* and hit enter. Type again *hello2()*. What happens? Now try typing *hs.hello2()*. What do you see? Check out all the hello-functions. If you change something in your module 'hellos.py' but it does not seem to take effect, you need to re-import the module. Type *import importlib* to get the necessary python module, then type *importlib.reload(hs)*. See the file 'Figs4Paper.py' for examples. Alternatively, you could also exit from ipython and re-launch ipython.

e) Now move to the CE. Science context. First play a bit with the CE. In the file Figs4Paper.py, play with the different default options. Set the flags for different tests / figures to 'yes' instead of 'no'. Look at the plots you get, think a bit about them, about the CE and what it can (not) tell you.

f) Back to python. You should have seen a series of pre-defined plots. Among them temperature plots that focus on the global mean temperature of the atmosphere (as this was one of the objectives of the paper for which the code was used). The goal now is to also plot the temperature of the ocean. Go to TestDefs.py, search for the PlotTDTemp function. You will see that it is hard wired to the temperature of the atmosphere (index 0 in array T_of_t). Add a new, similar function to TestDefs.py that allows you (the user) to plot either the temperature of the atmosphere or the temperature of the ocean (index 1 in array T_of_t) or both (in separate figures).

g) Still within the context of the CE, you should have seen a 2D map in task 3 (cartopy). The associated function is *PlotTempPat* in the file PatternScaling.py. If you look at the function definition, you see quite a number of arguments listed that are passed to the function, e.g. proj or

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

Figure 6: Simple example of class definition and usage, task 6. Figure and image taken from https://docs.python.org/3/tutorial/classes.html

addcont. Re-write the function such that instead of these explicit arguments it uses **kwargs (keyword / argument pairs). To get an idea of how this maybe done, look into the file ClimDICE.py and search for kwargs, used e.g. in the function ModCarbEmi.

h) If you feel up for more: write a function with which you can add more mammals to the pandas data frame from task 2, point c. Or also remove mammals from the data frame. Or remove a column of the data frame.

3.6 Task 6: classes

Classes provide a means of bundling data and functionality together. In the case of the CE, the data comprises information on the equilibrium state of the system or on the exchange rates of carbon among atmosphere and ocean. The functionality comprises, for example, methods for re-mapping emissions per year to arbitrary time steps (other than one year) or also methods that ascertain consistency of parameter changes (e.g. to ascertain that the total carbon mass is conserved numerically, that no carbon is lost due to numerics instead of physics). Or simply numerical integrators to advance the carbon cycle and the temperature equations with time. The file ClimDICE.py contains a class definition.

While classes are of key relevance for many applications, we do not go into any details here. The present task just looks into a very simple example of a class definition, shown in Figure 6[8]. Put this class definition in a file (Chiens.py) and use *import Chiens as ch* to have the class visible. Create (instantiate) a dog or two, *a=ch.Dog('Wuff')*, and characterize some of the tricks it knows, *a.add_trick("sit")*. Check the result via *a.tricks* and *a.name*. Add other functions to the class. Maybe try adding properties of the dog, like color, weight etc. Or enable it to bark[9].

A few more remarks on the example. The class Dog serves as a 'blue print' to create concrete dogs (instances of the class), one dog (instance) at a time with its properties. Upon initialization (instantiation) of a concrete dog, the __init__ function of the class Dog is executed, leaving your concrete dog with some initial, personal properties. The 'self' refers to a concrete instance of the class Dog, to one concrete dog.

A variant of the script, Chiens2.py, illustrates the pitfalls of (implicity) shallow copy as compared to (explicit) deep copy in python. See the Chiens2.py file for details. See also Task 8 or the internet.

3.7 Task 7: team work

This tasks has two goals: repeat what you learned so far and train your skills in coding in a team, i.e., write clean, documented, and re-usable code. As a team, build a code to create, manipulate,

[8]The example is from https://docs.python.org/3/tutorial/classes.html

[9]Rough idea on how to do this: def bark(nbark): for ib in range(nbark): print("WUFF! ", end=") and then e.g. a.bark(3)

and visualize a pandas data frame for country specific data (population, Gross Domestic Product GDP, area, life expectancy, agricultural production, literacy, what else you want). One country per row, data in columns. Search the internet for data you find interesting. Plots may be maps with countries color coded according to data, or bar plots, line plots, scatter plots etc. Up to you.

A possible code structure is to have a python module containing functions AddRow2Df, AddCol2Df, AddVal2Df that allow you to add a row / column / value to a data frame. Likewise, you may think of functions MapPlotFromCol, BarPlotFromCol, ScatterFromCols etc. Then maybe have a python script where you import this module, set up / manipulate your data frame and produce some plots. Split the coding of the python script and python module (or however you wish to organize the task) among yourselves as you wish. Try to rely mostly on the code as such, not on bothering your team mate with (too many) questions.

Gather country specific data from the internet, e.g. GDP from Wikipedia. Get the data into your data frame via 'enter by hand' (using your functions) or write a function to read data (which you download from the web) from disk. You could also try to combine gridded data with shape files for countries (their borders) to calculate country averages from gridded data.

If you want, try to collaborate using a git-repository (e.g. gitlab, which is open source, or github, which belongs to the Microsoft universe). Associated knowledge may come in handy in your future, but as it is beyond the actual scope of the IPP, you would be on your own.

For hand-in: This is an advanced task. Therefore, it is not compelling that you hand-in something. Still, it is highly encouraged that you either hand-in something or, better yet, present some results (plots!) from this task on the last half day of the IP.

3.8 Task 8: more advanced topics

This final task contains further suggestions of what to do / try, but with less hints on how to do it.

a) Go back to Figs4Paper.py and PatternScaling.py. In Figs4Paper.py, instead of plotting only a map for the year 2080, implement a loop over all years from 1850 to 2100. Change the plotting function in PatternScaling.py such that the names of the png-files contain the year. Use python or some other software (ffmpeg, mencoder) to produce a movie from the maps you got in this way.

b) As in a), but produce figures with two subplots: one showing the map (as in a) the other showing e.g. the CO_2 concentration as function of time and where in time you are (e.g. moving marker or line growing with time. Again assemble all the plots into a movie.

c) In the function PatScalTemp in the file PatternScaling.py you see that there are different CMIP5 models available for pattern scaling. In Figs4Paper.py, add a loop over all these models and produce a gallery of maps (on map per model) of how the year 2080 under RCP85 may look like, depending on what model is used for pattern scaling. Complement this gallery with a gallery of (normalized) histograms showing as a function of warming (x-axis) the fraction of grid boxes (y-axis) undergoing this warming.

d) Instead of specifying the names of the models by hand, get them automatically. Make use of python modules *os, glob, re*. From within python, get a listing of 'PATTERN*' files in folder 'pattern_library' using something like this *files = sorted(glob.glob(os.path.join(folder,fileid)))* (appropriately adapted). Then use regular expressions (module re, re.split(), re.search() etc.) to extract the model names from the file names.

e) Dive into numpy arrays, into views versus copies. Likewise, dive into the topic of 'deepcopy'. In ClimDICE.py, you also find it is used (has to be used!). See e.g. these links:

https://scipy-cookbook.readthedocs.io/items/ViewsVsCopies.html
https://www.programiz.com/python-programming/shallow-deep-copy used!).

f) Use scaling patterns remapped to the same (coarse) lat-lon-grid (folder pattern_scaling_cg) to compare the patterns. Load them all, put them into a pandas data frame. Produce a map showing the range of the patterns; the maximum warming; the minimum warming; color the maximum (minimum) temperature change map by the model number (instead of coloring by temperature change).

References

Doris Folini, Felix Kübler, Aleksandra Malova, and Simon Scheidegger. The climate in climate economics. *The Review of Economic Studies, open copy under https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3885021*, 2024. doi: 10.1093/restud/rdae011.

Corinne Le Quéré, Robbie M. Andrew, Pierre Friedlingstein, Stephen Sitch, Judith Hauck, Julia Pongratz, Penelope A. Pickers, Jan Ivar Korsbakken, Glen P. Peters, Josep G. Canadell, Almut Arneth, Vivek K. Arora, Leticia Barbero, Ana Bastos, Laurent Bopp, Frédéric Chevallier, Louise P. Chini, Philippe Ciais, Scott C. Doney, Thanos Gkritzalis, Daniel S. Goll, Ian Harris, Vanessa Haverd, Forrest M. Hoffman, Mario Hoppema, Richard A. Houghton, George Hurtt, Tatiana Ilyina, Atul K. Jain, Truls Johannessen, Chris D. Jones, Etsushi Kato, Ralph F. Keeling, Kees Klein Goldewijk, Peter Landschützer, Nathalie Lefèvre, Sebastian Lienert, Zhu Liu, Danica Lombardozzi, Nicolas Metzl, David R. Munro, Julia E. M. S. Nabel, Shin-ichiro Nakaoka, Craig Neill, Are Olsen, Tsueno Ono, Prabir Patra, Anna Peregon, Wouter Peters, Philippe Peylin, Benjamin Pfeil, Denis Pierrot, Benjamin Poulter, Gregor Rehder, Laure Resplandy, Eddy Robertson, Matthias Rocher, Christian Rödenbeck, Ute Schuster, Jörg Schwinger, Roland Séférian, Ingunn Skjelvan, Tobias Steinhoff, Adrienne Sutton, Pieter P. Tans, Hanqin Tian, Bronte Tilbrook, Francesco N. Tubiello, Ingrid T. van der Laan-Luijkx, Guido R. van der Werf, Nicolas Viovy, Anthony P. Walker, Andrew J. Wiltshire, Rebecca Wright, Sönke Zaehle, and Bo Zheng. Global Carbon Budget 2018. *Earth System Science Data*, 10(4): 2141–2194, December 2018. doi: 10.5194/essd-10-2141-2018.