GEO3231PythonReferenceSheet

NOTE the bits highlighted in blue are the bits you can change (e.g. swap for some other data) - the rest is the fundamental part of the command. However, note that in some instances where the example has a few lines of code, if you change one of the variable names, you may have to change all instances of that variable name.

Importing modules, and what those modules do

Note, if you don't know which you need, just import them all!

from numpy import *

Imports the module bumpy, which turns Python into a numerical analysis tool (akin to Matlab form those of you who've used Matlab)

from matplotlib.pyplot import *

Imports the module containing the plotting tools

from iris import *

This imports all (* means all in computing termanology) the bits and pieces of a module (called 'iris') that allows us to interact with large observational and climate model datasets stored in a type of file called 'netcdf' (files with an extension (i.e. a file ending like .docx or .pdf) '.nc')

from iris.analysis import *

This imports the bits of the 'iris' module required to start playing around with this 'netcdf' data

import iris.quickplot as quickplot

This imports the tools used to plot simple maps etc. with data that has been read in using any of the 'iris' modules (see those above)

from iris.coord_categorisation import *

This reads in a module that allows us to do clever things like converting monthly data to yearly data

from iris.analysis.cartography import *

This brings in a clever mapping module which can work out from the latitudes and longitudes at the corners of each box, what the area of the box is and lots more...

from scipy.stats import *

Model to do advances statistical analysis

from scipy.stats.mstats import *

Model to do further advances statistical analysis (for example producing linear regression models)

Help (manual pages)

help(command_name)

e.g. help(range) - displays the help text for the common you put in the brackets

QUIT help by typing 'q'

Making an array with some data in

my_array = array([2,5,7,4,3,2,4,5,6,8])

Making an array that is automatically full of a sequence of numbers

result_variable = linspace(10,30,50)

This will give you an array with 50 values equally spaced between 10 and 30

Print

print 'hello'

does what is says...

Basic Maths

```
from numpy import *
my_array1 = array([2.0,5.0,7.0])
my_array2 = array([2.0,2.0,2.0])
c =  my_array1 + my_array2
print c
d =  my_array1 / my_array2
print d
e =  my_array1 * my_array2
print e
f =  my_array1 - my_array2
print f
```

Basic stats

```
from numpy import *
my_array1 = array([2.0,5.0,7.0])
c =  max(my_array1)
 print c
d =  min(my_array1)
print d

e =  mean(my_array1)

print e
```

For Loop

```
for counter in [1,2,3,4,5,6]:
    print counter
```

cycles through the list/array that you specify (here [1,2,3,4,5,6]) and performs whatever operation you specify in the indented section of the loop, on each element of the list/array sequentially

Plotting

`show()`

displayed whatever you gave plotted

line plot

`plot(variable1, variable2)`

where for example `variable1 = array([1,2,3,4])` and `variable2 = array([8,7,6,5])`

Scatter plot

`scatter(variable1, variable2)`

Adding labels and titles to plots

```
title('my plot')
xlabel('variable 1')
ylabel('variable 2')
```

Where xlabel and ylabel and the x and y axis titles respectively

Adding a legend to a plot

```
plot(variable_1,variable_2, label = 'max')
plot(variable_1,variable_3, label = 'min')
legend()
```

If you are plotting two lines, you might want a legend to explain what they are. Here we have those two lines and a legend where they are labelled 'max' and 'min'

Contour plot of raw data (i.e. not of a cube of data that contains all of the metadata as well as the numbers)

```
contourf(data)
```

Note that a contour (without the f produces a contour plot, but inly plots the contour lines rather than filling between the lines with the relevant colour)

So, for example, if you wanted a coloured contour map as the base map, then a line contour map (like altitude on an OS map) op top, do:

```
contourf(data1)

contour(data2)
```

Contour plot from an 'iris cube' i.e. a variable read in using one of the iris modules - a variable holding the (e.g.) climate data as well as all the metadata about that climate model etc.

```
quickplot.contourf(my_cube,31)
```

Note that the '31' here is how many colour levels you want to use - you can of course leave it blank

x-y plot from data that was read in as an iris cube then averaged so that it ply has a single dimension (i.e. it has been spatially averaged so the only dimension is time)

```
quickplot.plot(my_cube)
```

And as above, but specifying how many ticks (numbers) you want on the x-axis:

```
import matplotlib.pyplot as plt
import iris.quickplot as quickplot

number_of_xaxis_ticks = 5

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

quickplot.plot(my_cube)

xloc = plt.MaxNLocator(number_of_xaxis_ticks)
ax.xaxis.set_major_locator(xloc)

plt.show()
```

Adding coastlines to a map plot

```
gca().coastlines()
```

Coloured contour plot from an 'iris cube' with a line counter map (like altitude contours from an OS map) on top:

```
quickplot.contourf(my_cube_1,31)

quickplot.contour(my_cube_2,31) # i.e. no 'f' at the end of cotour
```

Where my_cube_1 and my_cube_2 are the two datasets you want to produce. e.g. here

Plotting a map of the Pacific (and not having a big white gap)

```
import iris.plot as iplt
import cartopy.crs as ccrs

west = -250
east = -100
south = -90
north = 90

temporary_cube = cube.intersection(longitude = (west, east))
my_regional_cube = temporary_cube.intersection(latitude = (south, north))

ax1 = plt.subplot(111,projection=ccrs.PlateCarree(central_longitude=np.round(west + (east - west)/2.0)))
my_plot = iplt.contourf(my_regional_cube)
plt.show()
```

Plotting a line plot with coloured bands showing uncertainties/standard deviations

example script

Reading data in from a text file

```
my_data = genfromtxt('filename',skip_header = 3)
```

Reads columns of data from a text file into an array, here called my_data. In this case it skips the first two lines assuming that this has column titles etc.

or if a cvs file:

```
my_data = genfromtxt('filename.csv',skip_header = 3,delimiter = ',')
```

delimiter specifies what character is separating the columns - typically a comma, but it could be a ';' or a tab (which is '\t') or a space ' ' etc.

Loading climate model or big observational data from netcdf files (files with the extension '.nc') using iris

```
my_cube = load_cube('filename')
```

Note that if this does not work and tells you ether are too many cubes, but can try:

```
my_cube = load('filename')
```

This is more flexible about how to reads in data, but you might find that you have read in various different things at one, so will want to check this after (print my_cube), then potentially extract what you want form my_cube

Converting monthly data to yearly data in an iris cube

```
add_year(my_monthly_cube, 'time', name='year')
my_new_annual_cube= my_monthly_cube.aggregated_by('year', MEAN)
```

Calculating an annual mean but just over specific months- from an iris cube

```
import iris
from iris.coord_categorisation import *

#extracting just two months from a cube with all of teh months - here 12 and 1: December and January
cube = iris.load_cube('your_monthly_cube_location_and_name.nc')
add_month_number(cube, 'time', name='month_number')
cube2 = cube[np.where((cube.coord('month_number').points == 12) | (cube.coord('month_number') == 1))]

#then to average this by each year, so that you have the December-Jan for each year add the 'season year', i.e. a number of each 'season'
add_season_year(cube2, 'time', name='season_year')

#then average by the season year:
cube2.aggregated_by(['season_year'], iris.analysis.MEAN)
```

#cube 2 then contains the averaged for each december-january period in each year

Averaging an iris cube of data along some dimension (typically across the time-intervals, or latitude etc.)

average_across_time = my_cube.collapsed(['time'],MEAN)

or when you want to average latitudinal and longitudinally and need to account for the different sizes of a (e.g.) 1x1 degree grid box on the equator verses a 1x1 degree grid box at the pole

my_cube.coord('latitude').guess_bounds()
my_cube.coord('longitude').guess_bounds() #These two lines are just something you sometimes have to do when the original data did not include all of the latitude/longitude data we require
grid_areas = area_weights(my_cube)
global_average_variable = my_cube.collapsed(['latitude', 'longitude'],MEAN, weights=grid_areas) #This does the clever maths to work out the grid box areas

Writing a script

see week 1 practical: Week 1 An introduction to scientific computing

Calculating the mean of an array

my_mean = mean(my_array)

Calculating the mean of an array which has 'nan' - Not A Number - 'gaps' in it

from scipy.stats import *
my_mean = nanmean(my_array)

Calculating the standard deviation of an array

my_std = std(my_array)

Calculating the standard deviation of an array which has 'NaN' - Not s Number - 'gaps' in it

from scipy.stats import *
my_std = nanstd(my_array)

Replacing a missing data indicator with a nan

from numpy import *

data = array([1.0,2.0,3.0,4.0,5.0,-99999.0,7.0,8.0,9.0,-99999.0])

#if we make a simply array containing some numbers and some 'missing data values', where -99999.0 - e.g. numbers to highlight where there were no observations, we can then change them to Not A Number (nan) values like so:

data[where(data == -99999)] = nan

print data

#If we then plotted this data it would commit these nan values

masked

Masking where you have NaNs in a dataset

```
import numpy.ma as ma
masked_data = ma.masked_invalid(data)
```

for more information about masks see the relevant numpy help pages

Calculating the correlation between two datasets

```
my_correlation_variable = spearmanr(variable_1,variable_2)
```

Linear Regression

```
from scipy.stats.mstats import *
slope, intercept, r_value, p_value, std_err = linregress(variable_1,variable_2)
```

Calculates the slope (m) and intercept (c) from a linear relationship (the equation of which is y = m*x + c), as well as the stats on that relationship (r-value, p-value, standard error etc.)

Multiple linear regression example

```
import numpy as np
import matplotlib.pyplot as plt

#Your y-value
y = [-6,-5,-10,-5,-8,-3,-6,-8,-8]

#Your 3 x-values (note that you could have more/less but would have to edit the subsequent code)
x1 = [-4.95,-4.55,-10.96,-1.08,-6.52,-0.81,-7.01,-4.46,-11.54]
x2 = [-5.87,-4.52,-11.64,-3.36,-7.45,-2.36,-7.33,-7.65,-10.03]
x3 = [-0.76,-0.71,-0.98,0.75,-0.86,-0.50,-0.33,-0.94,-1.03]

#combines the x's into one variable
x = [x1,x2,x3]

#does the least squares fitting
X = np.column_stack(x+[[1]*len(x[0])])
m3,m2,m1,c = np.linalg.lstsq(X,y)[0]

#plots the result. The original y is in red, and the multiple linear regression estimates values in green
plt.plot(y,'r')
plt.plot(m3*x3 + m2*x2 + m1*x1 + c,'g–')
plt.show()
```

Plotting a linear best fit line

```
slope, intercept, r_value, p_value, std_err = linregress(variable_1,variable_2)

scatter(variable_1,variable_2)
x = variable1
y = slope*variable1+intercept
plot(x,y)
show()
```

Calculating seawater density

```
import seawater
results_array = seawater.dens(salinity_data,temperature_data,1)
```

Note the value '1' at the end is saying there is a pressure of 1. If you were calculating density for deep in the ocean, technically you should account for the increased pressure of seawater. Type help(seawater.dens) to find out more about this

Working with Iris Cubes (i.e. data read in from netcdf files using the iris modules)

Contour plot from an 'iris cube' i.e. a variable read in using one of the iris modules - a variable holding the (e.g.) climate data as well as all the metadata about that climate model etc.

```
quickplot.contourf(my_cube,31)
```

Note that the '31' here is how many colour levels you want to use - you can of course leave it blank

x-y plot from data that was read in as an iris cube then averaged so that it ply has a single dimension (i.e. it has been spatially averaged so the only dimension is time)

```
quickplot.plot(my_cube)
```

As above, but specifying that you want a specific dimension (here depth) on the y-axis:

```
quickplot.plot(my_data,my_cube.coord('depth'))
show()
```

Adding coastlines to a map plot

```
gca().coastlines()
```

Loading climate model or big observational data from netcdf files (files with the extension '.nc') using iris

```
my_cube = load_cube('filename')
```

Note that if this does not work and tells you ether are too many cubes, but can try:

```
my_cube = load('filename')
```

This is more flexible about how to reads in data, but you might find that you have read in various different things at one, so will want to check this after (print my_cube), then potentially extract what you want form my_cube

Converting monthly data to yearly data in an iris cube

```
add_year(my_monthly_cube, 'time', name='year')
```

```
my_new_annual_cube= my_monthly_cube.aggregated_by('year', MEAN)
```

Averaging an iris cube of data along some dimension (typically across the time-intervals, or latitude etc.)

```
average_across_time = my_cube.collapsed(['time'],MEAN)
```

or when you want to average latitudinal and longitudinally and need to account for the different sizes of a (e.g.) 1x1 degree grid box on the equator verses a 1x1 degree grid box at the pole

```
my_cube.coord('latitude').guess_bounds()
my_cube.coord('longitude').guess_bounds() #These two lines are just something you sometimes have to do when the original data did not include all of the latitude/longitude data we require
grid_areas = area_weights(my_cube)
global_average_variable = my_cube.collapsed(['latitude', 'longitude'],MEAN, weights=grid_areas) #This does the clever maths to work out the grid box areas
```

Extracting horizontal slices from a cube (e.g. at a given depth)

```
my_depth_slice_variable = my_cube.extract(Constraint(depth = 0))
```

This example extracts the surface level from a 3D cube

Extracting vertical slices from a cube

```
my_meridional_slice_variable = my_cube.extract(Constraint(longitude = 182.5))
```

Of course here 'longitude' could be (in most situations) replaced by latitude, depth of time - although you would have to specify a different value because (for example) 182.5 does not mean anything sensible as a latitude

Averaging together all of the latitudes - i.e. like a depth slice, but averaged over the region of your dataset

```
variable_collapsed_along_longitude = my_cube.collapsed('longitude',MEAN)
```

Extracting a spatial region from a global dataset

```
west = -70
east = 50
south = -20
north = 60
```

```
temporary_cube = my_cube.intersection(longitude = (west, east))
my_regional_cube = temporary_cube.intersection(latitude = (south, north))
```

> note that while I've left the variables 'west' 'east' etc. red, there is no reason why these need to have those names, bit I thought it easier to see what this was doing this way

Extracting the data part of the iris cube (rather than the metadata)

```
my_data = my_cube.data
```

Getting the year (or month or day) values for a cube of data

> note if you just look at the time coordinate it is usually something crazy like 'number of days since 1st Jan 1850'

```
coord = my_cube.coord('time')
date_variable = array([coord.units.num2date(value).year for value in coord.points])
```

Correlation maps

```
import iris.analysis.stats as istats
my_correlation_variable = istats.pearsonr(my_cube_1,my_cube_2,corr_coords=['time'])
```

Calculating the difference point-by-point between two cubes of data

```
my_difference_cube = my_cube_1 - my_cube_2
```

Calculating the mean of a bunch of cubes

```
my_mean_variable = mean([my_cube_1,my_cube_2,my_cube_3])
```

Calculating the variance of a bunch of cubes

```
my_mean_variable = var([my_cube_1,my_cube_2,my_cube_3])
```

Doing the carbonate-chemistry calculations

```
import carbchem_modified
```

```
co2_cube =
carbchem_modified.carbchem(1,temperature_cube.data.fill_value,temperature_cube,salinity_cube,dissolved_carbon_cube,alkalinity_cube)
```

> There is actually a bit more flexibility here than the colours make out... change the one to a 1 and you can calculate pH for example - check out the help (assuming I wrote them) for more details.
> Also see the week 8 practical form more details on this: GEO3231Week8Practical

Plotting two time-series (or similar) datasets on the same x-axis but with different y axes:

```
timeseries_1 = [0.7135553808947991, 0.4861775160058641, 0.8254392727351161, 0.5098580281973034, 0.13077277305355084, 0.7123233371092856, 0.13198748832181673, 0.8517834038288471, 0.014955556391444969, 0.39952309433852173]
```

```
timeseries_2 = [ 90.01896882,  74.23638798,  76.70026728,  29.56024534, 6.43137557,  19.57561375,  91.57594645,  21.33809188,  66.18203436,  68.80092629]
```

We can plot then both on one graph, but because the values in timeseries_2 are much bigger we will not be able to visually compare any variation between the two time series because the variability in the small numbers will be invisible. e.g.:

```
plt.plot(timeseries_1)
plt.plot(timeseries_2)
plt.show()
```

produces this

```
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()
# this tells python that you will want a figure and it will have one set of axes called 'ax1'

ax1.plot(timeseries_1,'red')
#This plots data on that axis

ax2 = ax1.twinx()
#This tells python that  you want a second set of axes for a different dataset. We want to share (twin) the x-axes between the two different set of axes this is what twinx() does. The new set of axes is called 'ax2'

ax2.plot(timeseries_2,'blue')
#Now we plot some data on this second set of axes.

 plt.show()
```

produces this

Saving a figure you have plotted:

```
from import matplotlib.ptplot import *

timeseries_1 = [0.7135553808947991, 0.4861775160058641, 0.8254392727351161, 0.5098580281973034, 0.13077277305355084, 0.7123233371092856, 0.13198748832181673, 0.8517834038288471, 0.014955556391444969, 0.39952309433852173]

timeseries_2 = [ 90.01896882,  74.23638798,  76.70026728,  29.56024534, 6.43137557,  19.57561375,  91.57594645,  21.33809188,  66.18203436,  68.80092629]

plot(timeseries_1,timeseries_2)

savefig('/my_directory/my_filename.png')
```

where /my_directory/my_filename.png is specfying the location and name of the file you want to save

Regridding and doing initial processing of non-CMIP5 data. Example here is SODA data (not python but related)

Your files may have a number of different variables in them (e.g. temperature and sailinity). We therefore first want to see what is in the files. Logging on to atoll (and possibly not logging from that to a node) type the following. NOTE YOU ARE TYPING THIS INTO THE COMMAND LINE WITHOUT 1st OPENING PYTHON! This is not to be done in python.

ncdump -h file_name.nc (where file_name.nc is the name of your file). This will print out something like this. Here you can see what the variables are. It also tells you what the code is for each variable, for example find where it says 'TEMPERATURE', on the line above that you see that this is a variable called 'temp' with depth, latitude and longitude information. Above 'SALINITY' you see you have a variable called 'salt' etc... So say you want to look at temperature, the variable name you are after is 'temp'

We can then make a new file that holds just that temperature (temp) variable by typing:

```
cdo selname,temp file_name.nc new_file.nc
```

Now just the temperature data is in the file called 'new_file.nc'

If you have a number of files (e.g. each monthly file of the SODA dataset) you will first want to combine these in to one file:

```
cdo mergetime file_name_1.nc file_name_2.nc my_output_file.nc
```

You can cheat if you want to merge together all of the '.nc' files in a directory and type cdo mergetime *.nc my_output_file.nc

You can then regard the dataset onto a 360 by 180 degree (one by one degree) grid (or other grid if you choose) with:

```
cdo remapbil,r360x180 my_input_file.nc my_output_file.nc
```

This takes whatever is in 'my_input_file.nc' and puts it on a nice simple one by one grid in 'my_output_file.nc'. Note if you were to change the 360 or 180 degree numbers you would end up with a different number of latitude and longitude points.

Your data is now nicely processed and ready to analyse exactly as the various tutorials explain.

Reading numerical data in from a text file:

data = np.genfromtxt('input_data_file.txt')

column1_data = data[:,0]

column2_data = data[:,1]

Saving data to a text file:

np.savetxt('output_data_file.txt',np.c([column1_data,column2_data])

A function to create running means (smoothing data):

To use this, just copy the code below into your Python window, then use as any other function.

```
import numpy as np

def running_mean(x, N):
        y = np.zeros((len(x),))
        for ctr in range(len(x)):
                y[ctr] = np.sum(x[(ctr-np.round(N/2)):ctr+np.round(N/2)])
        out = y/N
        out[0:np.round(N/2)] = np.NAN
        out[-1.0*np.round(N/2)::] = np.NAN
        return out
```

Here x is the dataset and N the smoothing window (the number of datapoint you want to average together to create your smoothing)

Adding latitude and longitude lables to your map

http://scitools.org.uk/cartopy/docs/latest/examples/tick_labels.py

Calculating the overturning stream function example (using the CMIP5 msftmyz variable):

```
import numpy as np
import matplotlib.pyplot as plt
import iris

#Read in cube
cube = iris.load_cube('MPI-ESM-P_msftmyz_piControl_regridded.nc')

#work out what out latitude coordinate is called
print cube.coords

#We see that the latitude coordinate is called something like grid_latitude. Make a note of this.
```

```
#Using the name identified above, we can print out all of the latitude points
print cube.coord('grid_latitude').points

#we can then find the position of the latitude of interest - probably that closes to 26N. E.g. it
cube.coord('grid_latitude').points[116]

#Make a new cube (called cube2) with just depth and time for that latitude
cube2 = cube[:,0,:,116]

#Make an array to hold the values of all of the stream function values - i.e. an array the same s
my_MOC_array = np.zeros(np.shape(cube)[0])

#Then produce a loop to pull out the maximum values from this latitude for all times.
for i,slice in enumerate(cube2.slices(['depth'])):
   my_MOC_array[i] = np.max(slice.data)


#Plot to make sure the data makes sense:
plt.plot(my_MOC_array)
plt.show()
```

Extracting a range of years from a cube

```
import numpy as np
coord = my_cube.coord('time')
date_variable = np.array([coord.units.num2date(value).year for value in coord.points])
years_of_interest = np.where((date_variable <= 2100) & (date_variable >= 2080))
my_cube = my_cube[years_of_interest]
```

Color maps:

Accent, Accent_r, Blues, Blues_r, BrBG, BrBG_r, BuGn, BuGn_r, BuPu, BuPu_r, CMRmap, CMRmap_r, Dark2, Dark2_r, GnBu, GnBu_r, Greens, Greens_r, Greys, Greys_r, OrRd, OrRd_r, Oranges, Oranges_r, PRGn, PRGn_r, Paired, Paired_r, Pastel1, Pastel1_r, Pastel2, Pastel2_r, PiYG, PiYG_r, PuBu, PuBuGn, PuBuGn_r, PuBu_r, PuOr, PuOr_r, PuRd, PuRd_r, Purples, Purples_r, RdBu, RdBu_r, RdGy, RdGy_r, RdPu, RdPu_r, RdYlBu, RdYlBu_r, RdYlGn, RdYlGn_r, Reds, Reds_r, Set1, Set1_r, Set2, Set2_r, Set3, Set3_r, Spectral, Spectral_r, Wistia, Wistia_r, YlGn, YlGnBu, YlGnBu_r, YlGn_r, YlOrBr, YlOrBr_r, YlOrRd, YlOrRd_r, afmhot, afmhot_r, autumn, autumn_r, binary, binary_r, bone, bone_r, brewer_Accent_08, brewer_Blues_09, brewer_BrBG_11, brewer_BuGn_09, brewer_BuPu_09, brewer_Dark2_08, brewer_GnBu_09, brewer_Greens_09, brewer_Greys_09, brewer_OrRd_09, brewer_Oranges_09, brewer_PRGn_11, brewer_Paired_12, brewer_Pastel1_09, brewer_Pastel2_08, brewer_PiYG_11, brewer_PuBuGn_09, brewer_PuBu_09, brewer_PuOr_11, brewer_PuRd_09, brewer_Purples_09, brewer_RdBu_11, brewer_RdGy_11, brewer_RdPu_09, brewer_RdYlBu_11, brewer_RdYlGn_11, brewer_Reds_09, brewer_Set1_09, brewer_Set2_08, brewer_Set3_12, brewer_Spectral_11, brewer_YlGnBu_09, brewer_YlGn_09, brewer_YlOrBr_09, brewer_YlOrRd_09, brg, brg_r, bwr, bwr_r, cool, cool_r, coolwarm, coolwarm_r, copper, copper_r, cubehelix, cubehelix_r, flag, flag_r, gist_earth, gist_earth_r, gist_gray, gist_gray_r, gist_heat, gist_heat_r, gist_ncar, gist_ncar_r, gist_rainbow, gist_rainbow_r, gist_stern, gist_stern_r, gist_yarg, gist_yarg_r, gnuplot, gnuplot2, gnuplot2_r, gnuplot_r, gray, gray_r, hot, hot_r, hsv, hsv_r, inferno, inferno_r, jet, jet_r, magma, magma_r, nipy_spectral, nipy_spectral_r, ocean, ocean_r, pink, pink_r, plasma, plasma_r, prism, prism_r, rainbow, rainbow_r, seismic, seismic_r, spectral, spectral_r, spring, spring_r, summer, summer_r, terrain, terrain_r, viridis, viridis_r, winter, winter_r

Producing pretty plots

example file

or copy of script here

High, low and band-pass filtering cubes

```
import iris
import matplotlib.pyplot as plt
import scipy
import scipy.signal
import iris.quickplot as qplt
import numpy as np


'''
We will use the following functions, so make sure they are available
'''

def butter_bandpass(lowcut,  cutoff):
    order = 2
    low = 1/lowcut
    b, a = scipy.signal.butter(order, low , btype=cutoff,analog = False)
    return b, a

def low_pass_filter(cube,limit_years):
        b1, a1 = butter_bandpass(limit_years, 'low')
        output = scipy.signal.filtfilt(b1, a1, cube,axis = 0)
        return output

def high_pass_filter(cube,limit_years):
        b1, a1 = butter_bandpass(limit_years, 'high')
        output = scipy.signal.filtfilt(b1, a1, cube,axis = 0)
        return output


'''
Initially just reading in a dataset to work with, and averaging lats and longs to give us a times
'''

file = '/media/usb_external1/cmip5/tas_regridded/MPI-ESM-P_tas_piControl_regridded.nc'
cube = iris.load_cube(file)

timeseries1 = cube.collapsed(['latitude','longitude'],iris.analysis.MEAN)

'''
Filtering out everything happening on timescales shorter than than X years (where x is called low
'''

lower_limit_years = 10.0
output_cube = cube.copy()
output_cube.data = low_pass_filter(cube.data,lower_limit_years)

timeseries2 = output_cube.collapsed(['latitude','longitude'],iris.analysis.MEAN)
```

```
plt.close('all')
qplt.plot(timeseries1 - np.mean(timeseries1.data),'r',alpha = 0.5,linewidth = 2)
qplt.plot(timeseries2 - np.mean(timeseries2.data),'g',alpha = 0.5,linewidth = 2)
plt.show(block = True)

'''
Filtering out everything happening on timescales longer than than X years (where x is called uppe
'''

upper_limit_years = 5.0
output_cube = cube.copy()
output_cube.data = high_pass_filter(cube.data,upper_limit_years)

timeseries3 = output_cube.collapsed(['latitude','longitude'],iris.analysis.MEAN)

plt.close('all')
qplt.plot(timeseries1 - np.mean(timeseries1.data),'r',alpha = 0.5,linewidth = 2)
qplt.plot(timeseries3 - np.mean(timeseries3.data),'b',alpha = 0.5,linewidth = 2)
plt.show(block = True)

'''
Filtering out everything happening on timescales longer than than X years (where x is called uppe
'''

upper_limit_years = 50.0
output_cube = cube.copy()
output_cube.data = high_pass_filter(cube.data,upper_limit_years)
lower_limit_years = 5.0
output_cube.data = low_pass_filter(output_cube.data,lower_limit_years)

timeseries4 = output_cube.collapsed(['latitude','longitude'],iris.analysis.MEAN)

plt.close('all')
qplt.plot(timeseries1 - np.mean(timeseries1.data),'r',alpha = 0.5,linewidth = 2)
qplt.plot(timeseries4 - np.mean(timeseries4.data),'y',alpha = 0.5,linewidth = 2)
plt.show(block = True)


'''
Hopefully this tells you everything you need. Just be aware  that strange things can happen at he
'''
```

Calculating the depths at which a variable in a cube moved from above to below a critical value

for example, for calculating the saturation horizon, or the depth of a isopycnal

```
import iris
import numpy as np
```

```
import matplotlib.pyplot as plt
import unicodedata

directory = '/data/NAS-ph290/ph290/cmip5/last1000/' #the location of the directory holding the in
file = 'HadCM3_thetao_past1000_r1i1p1_regridded_not_vertically_Omon.nc' #the name of the input fi
critical_value = 1.0 # i.e. identify the depth for which values move from above to below this val

#load the cube in from which you want to calculate the depth data
cube = iris.load_cube(directory + file)

#This script is a little more complicated than you might expect, so that it can work with cubes o
shape = np.shape(cube)
test = np.size(shape)

if test == 4:
    print 'cube has 4 dimensions, assuming time, depth, latitude, longitude'
    depth_coord = 1
elif test == 3:
    print 'cube has 3 dimensions, assuming depth, latitude, longitude'
    depth_coord = 0
else:
    print 'are you sure cube has a depth dimension?'


#Identify the values for each depth level
depths = cube.coord(dimensions = depth_coord).points
#Make a cube that just holds the depth data
depths_cube = cube.copy()
if test == 4:
    depths_cube.data = np.ma.masked_array(np.swapaxes(np.tile(depths, (shape[0],shape[3], shape[2
elif test == 3:
    depths_cube.data = np.ma.masked_array(np.swapaxes(np.tile(depths, (shape[2],shape[1],1)),2,0)


depths_cube.data.mask = cube.data.mask

#mask that cube were depths are less than your chosen value
mask = np.where(cube.data <= critical_value)
depths_cube.data.mask[mask] = True

#make a cube without a depth dimension to hold the output
depth_name = cube.coord(dimensions = depth_coord).standard_name.encode('ascii','ignore')
output_cube = cube.collapsed(depth_name,iris.analysis.MEAN)

#Find the greatest depth at which data is found above the value of interest
if test == 4:
    output_cube.data = np.max(depths_cube.data,axis = 1)
elif test == 3:
    output_cube.data = np.max(depths_cube.data,axis = 0)


#Give the cube the right metadata
output_cube.standard_name = 'depth'
output_cube.units = 'm'
```

```
#output_cube contains the depths
```

Calculating the NINO3.4 index from a cube

```
#####
# NOTE, this is a bit of a fudge at present - revisit and improve
#####

import iris
import iris.coord_categorisation
import numpy as np

print 'please note, this script has not yet been tested with observations, so perform your own te

#Reading in the MONTHLY SST data from the netcdf file
cube = iris.load_cube('/data/NAS-ph290/ph290/cmip5/historical/tos_Omon_HadCM3_historical_r1i1p1_1
iris.coord_categorisation.add_season_year(cube, 'time', name='season_year')
iris.coord_categorisation.add_month_number(cube, 'time', name='month_number')
cube = cube.aggregated_by(['season_year','month_number'], iris.analysis.MEAN)

coord = cube.coord('time')
dt = coord.units.num2date(coord.points)

years = np.array([coord.units.num2date(value).year for value in coord.points])
lon_west = -170
lon_east = -120
lat_south = -0.5
lat_north = 0.5

cube_region_tmp = cube.intersection(longitude=(lon_west, lon_east))
cube_region = cube_region_tmp.intersection(latitude=(lat_south, lat_north))

timeseries = cube_region.collapsed(['latitude','longitude'],iris.analysis.MEAN).data

years_1 = years[np.where(cube.coord('month_number').points == 1)]
timeseries_1 = timeseries[np.where(cube.coord('month_number').points == 1)]
years_2 = years[np.where(cube.coord('month_number').points == 2)]
timeseries_2 = timeseries[np.where(cube.coord('month_number').points == 2)]
years_3 = years[np.where(cube.coord('month_number').points == 3)]
timeseries_3 = timeseries[np.where(cube.coord('month_number').points == 3)]
years_4 = years[np.where(cube.coord('month_number').points == 4)]
timeseries_4 = timeseries[np.where(cube.coord('month_number').points == 4)]
years_5 = years[np.where(cube.coord('month_number').points == 5)]
timeseries_5 = timeseries[np.where(cube.coord('month_number').points == 5)]
years_6 = years[np.where(cube.coord('month_number').points == 6)]
timeseries_6 = timeseries[np.where(cube.coord('month_number').points == 6)]
years_7 = years[np.where(cube.coord('month_number').points == 7)]
timeseries_7 = timeseries[np.where(cube.coord('month_number').points == 7)]
years_8 = years[np.where(cube.coord('month_number').points == 8)]
timeseries_8 = timeseries[np.where(cube.coord('month_number').points == 8)]
```

```
years_9 = years[np.where(cube.coord('month_number').points == 9)]
timeseries_9 = timeseries[np.where(cube.coord('month_number').points == 9)]
years_10 = years[np.where(cube.coord('month_number').points == 10)]
timeseries_10 = timeseries[np.where(cube.coord('month_number').points == 10)]
years_11 = years[np.where(cube.coord('month_number').points == 11)]
timeseries_11 = timeseries[np.where(cube.coord('month_number').points == 11)]
years_12 = years[np.where(cube.coord('month_number').points == 12)]
timeseries_12 = timeseries[np.where(cube.coord('month_number').points == 12)]


thirty_yr_means = {}
thirty_yr_means['1'] = []
thirty_yr_means['2'] = []
thirty_yr_means['3'] = []
thirty_yr_means['4'] = []
thirty_yr_means['5'] = []
thirty_yr_means['6'] = []
thirty_yr_means['7'] = []
thirty_yr_means['8'] = []
thirty_yr_means['9'] = []
thirty_yr_means['10'] = []
thirty_yr_means['11'] = []
thirty_yr_means['12'] = []

meaning_years = {}
meaning_years['1'] = []
meaning_years['2'] = []
meaning_years['3'] = []
meaning_years['4'] = []
meaning_years['5'] = []
meaning_years['6'] = []
meaning_years['7'] = []
meaning_years['8'] = []
meaning_years['9'] = []
meaning_years['10'] = []
meaning_years['11'] = []
meaning_years['12'] = []

for j in np.arange(12)+1:
        if j == 0:
                years_tmp = years_0
                timeseries_tmp = timeseries_0
        if j == 1:
                years_tmp = years_1
                timeseries_tmp = timeseries_1
        if j == 2:
                years_tmp = years_2
                timeseries_tmp = timeseries_2
        if j == 3:
                years_tmp = years_3
                timeseries_tmp = timeseries_3
        if j == 4:
                years_tmp = years_4
                timeseries_tmp = timeseries_4
```

```
        if j == 5:
                years_tmp = years_5
                timeseries_tmp = timeseries_5
        if j == 6:
                years_tmp = years_6
                timeseries_tmp = timeseries_6
        if j == 7:
                years_tmp = years_7
                timeseries_tmp = timeseries_7
        if j == 8:
                years_tmp = years_8
                timeseries_tmp = timeseries_8
        if j == 9:
                years_tmp = years_9
                timeseries_tmp = timeseries_9
        if j == 10:
                years_tmp = years_10
                timeseries_tmp = timeseries_10
        if j == 11:
                years_tmp = years_11
                timeseries_tmp = timeseries_11
        for i in years_tmp[::5]:
                meaning_years[str(j)].append(i)
                loc = np.where(years_tmp == i)[0][0]
                try:
                        thirty_yr_means[str(j)].append(np.mean(timeseries_tmp[loc-30:loc+30]))
                except:
                        thirty_yr_means[str(j)].append(np.NAN)

nino34 = []
for i,timeseries_value in enumerate(timeseries):
        meaning_years_tmp = meaning_years[str(cube.coord('month_number').points[i])]
        thirty_yr_means_tmp = thirty_yr_means[str(cube.coord('month_number').points[i])]
        loc = np.searchsorted(meaning_years_tmp, years[i], side="left")
        if loc >= np.size(meaning_years_tmp):
                loc = np.size(meaning_years_tmp) -1
        if meaning_years_tmp[loc] == np.NAN:
                nino34.append(np.NAN)
        else:
                nino34.append(timeseries_value - thirty_yr_means_tmp[loc])



nino34 = np.array(nino34)
```

Extracting contour lines from a cube of data and saving them out as a shape file for ARC GIS

```
import iris
import matplotlib.pyplot  as plt
import numpy as np
import shapefile
```

```
#function to extract the coordinates from the contour lines
def get_contour_verts(cn):
    contours = []
    # for each contour line
    for cc in cn.collections:
        paths = []
        # for each separate section of the contour line
        for pp in cc.get_paths():
            xy = []
            # for each segment of that section
            for vv in pp.iter_segments():
                xy.append(vv[0])
            paths.append(np.vstack(xy))
        contours.append(paths)
    return contours

#Input file (if you need to read data in). Replace with your chosen file and directory
my_file = '/data/NAS-ph290/ph290/cmip5/historical/regridded/CCSM4_tos_historical_r1i1p1_regriddec


#Read the data in (note, here I'm adding [0] to only read in the first year of data)
cube = iris.load_cube(my_file)[0]

#specify the number of contour levels you want
no_contours = 20
cn = plt.contour(cube.data,no_contours)
contours = get_contour_verts(cn)

#write the coordinates to a shapefile
w = shapefile.Writer()
for cont in contours:
    w.poly(shapeType=3, parts=cont)



#enter the filename here:
filename = '/home/ph290/Downloads/my_shapefile.shp'
w.save(filename)
```

Cube Statistics/Maths

Finding the maximum value in latitude-longitude space:

import iris

import iris.analysis

my_result = my_cube.collapsed(['longitude','latitude'], iris.analysis.MAX)

Finding the minimum value in latitude-longitude space:

import iris

import iris.analysis

my_result = my_cube.collapsed(['longitude','latitude'], iris.analysis.MIN)

Other operations similar to MIN and MIN can be found here, and substituted in for where MAX and MIN are use din the above two examples

Adding a value (5 in this example) to the data in a cube

my_cube = my_cube + 5.0

Dividing the data in a cube by a certain value(5 in this example)

my_cube = my_cube / 5.0

Reading and concatenating EN4 data

EN4 data is a bit messy. This reads it in for you.

```
def my_callback(cube, field,files_tmp):
    # there are some bad attributes in the NetCDF files which make the data incompatible for merg
    cube.attributes.pop('history')
    cube.attributes.pop('creation_date')
    cube.attributes.pop('time', None)
    # if np.size(cube) > 1:
    #     cube = iris.experimental.concatenate.concatenate(cube)
    return cube


import glob
import iris

files = glob.glob('/home/cns205/data/EN.4.2.0.f.analysis.g10.2013*.nc')

cubes = iris.load(files,'sea_water_potential_temperature',callback=my_callback)
iris.util.unify_time_units(cubes)

for i in range(0,len(files)):
    del cubes[i].dim_coords[0].attributes['time_origin']


cube = cubes.concatenate_cube()
```

Identifying the grid coordinates in a cube of certain latitude and longitude points:

```
def find_lat_lon(cube,longitude,latitude):
    lon = cube.coord('longitude').points.copy()
    lat = cube.coord('latitude').points.copy()
    if np.max(lon) > 350:
        lon -= 180.0
    if np.max(lat) > 170:
        lat -= 90.0
    print 'required longitude grid box is: ',np.where(lon > longitude)[0][0]
    print 'required latitude grid box is: ',np.where(lat > latitude)[0][0]




#Edit the three lines below to fit what you want, then copy and paste the whole thing in to pytho
cube = MIROC5_SOIL_DJF_1990_2012
longitude = -22
latitude = -16

find_lat_lon(cube,longitude,latitude)
```

or

```
def find_lat_lon(cube,longitude,latitude):
    lon = cube.coord('longitude').points.copy()
    lat = cube.coord('latitude').points.copy()
    if np.max(lat) > 170:
        lat -= 90.0
    print 'required longitude grid box is: ',np.where(lon > longitude)[0][0]
    print 'required latitude grid box is: ',np.where(lat > latitude)[0][0]




#Edit the three lines below to fit what you want, then copy and paste the whole thing in to pytho
cube = MIROC5_SOIL_DJF_1990_2012
longitude = -22
latitude = -16

find_lat_lon(cube,longitude,latitude)
```

Lead/Lagged correlation e.g.

```
import numpy as np
from scipy.stats import *

x = np.random.rand(30)
```

```
y = np.roll(x,3)

max_lag = -5
max_lead = 5

for lag_tmp in range(max_lag,max_lead+1):
        tmp_x = np.roll(x,lag_tmp)
        my_correlation_variable = spearmanr(tmp_x,y)
        plt.scatter(lag_tmp,my_correlation_variable[0])


plt.show()
```

Hovmuller plots

```
import iris
import numpy as np
from iris.analysis import *
from iris.coord_categorisation import *
import iris.quickplot as quickplot
from matplotlib.pyplot import *

cube = iris.load_cube('/data/NAS-ph290/ph290/cmip5/last1000/HadCM3_sos_past1000_r1i1p1_regridded_
add_year(cube, 'time', name='year')

west = -30
east = 0
south = 0
north = 90
temporary_cube = cube.intersection(longitude = (west, east))
my_regional_cube = temporary_cube.intersection(latitude = (south, north))

average_across_time = my_regional_cube.collapsed(['longitude'],MEAN)

years = my_regional_cube.coord('year').points
lats = my_regional_cube.coord('latitude').points

close('all')
CS = contourf(lats,years,average_across_time.data,30)
cbar = colorbar(CS)
show()
```

and as anomalies

```
import iris
import numpy as np
from iris.analysis import *
from iris.coord_categorisation import *
import iris.quickplot as quickplot
from matplotlib.pyplot import *
```

```
cube = iris.load_cube('/data/NAS-ph290/ph290/cmip5/last1000/HadCM3_sos_past1000_r1i1p1_regridded_
add_year(cube, 'time', name='year')

west = -30
east = 0
south = 0
north = 90
temporary_cube = cube.intersection(longitude = (west, east))
my_regional_cube = temporary_cube.intersection(latitude = (south, north))
average_across_time = my_regional_cube.collapsed(['longitude'],MEAN)
average_across_time2 = average_across_time.collapsed(['time'],MEAN)

data = average_across_time.data
shape = np.shape(data)
for i in range(shape[0]):
    data[i,:] -= average_across_time2.data


years = my_regional_cube.coord('year').points
lats = my_regional_cube.coord('latitude').points


close('all')
CS = contourf(lats,years,data,30)
cbar = colorbar(CS)
show()
```

Drawing rectangles on plots:

http://matthiaseisen.com/pp/patterns/p0203/

Changing font size

```
from matplotlib.pyplot import *
import matplotlib

font = {'family' : 'normal',
        'weight' : 'bold',
        'size'   : 22}

matplotlib.rc('font', **font)
matplotlib.rcParams.update({'font.size': 22})

x = [1,2,3,4,5]
y = [6,7,8,9,10]

plot(x,y)
ylabel('y-value')
tight_layout()
show()
```

Contour plot with colour bar with title

```
import iris
import iris.plot as iplt
import matplotlib.pyplot as plt

cube = iris.load_cube('my_directory/my_cube.nc')
CS = iplt.contourf(cube[0])

cbar = plt.colorbar(CS,orientation = 'horizontal')
cbar.ax.set_title('my colour bar')

plt.show()
```

Plotting GLODAP profiles

```
import iris
import matplotlib.pyplot as plt
import iris.quickplot as qplt
import iris.analysis

cube_tmp = iris.load_cube('GLODAPv2.2016b.TAlk.nc','seawater alkalinity expressed as mole equival

cube =  iris.load_cube('/data/NAS-ph290/ph290/misc_data/temperature_annual_1deg.nc','sea_water_te

cube.data = cube_tmp.data

cube.coord('latitude').guess_bounds()
cube.coord('longitude').guess_bounds() #These two lines are just something you sometimes have to

grid_areas = iris.analysis.cartography.area_weights(cube)

global_average_variable = cube.collapsed(['latitude', 'longitude'],iris.analysis.MEAN, weights=gr

plt.plot(global_average_variable.coord('depth').points,global_average_variable.data) #plotting gl

plt.show()
```

Like Be the first to like this No labels