PYTHON FOR SCIENTIFIC COMPUTING

In [ ]:

OUTLINE

1. Python Background
2. Why Learn Python
3. Comments
4. Variables
5. Data Types
6. Python Operators
7. Conditional Statement
8. Loops
9. Functions
10. Data Structures
11. Modules
12. Pandas
13. Numpy
14. Matplotlib
15. Xarray
16. Cartopy

In [ ]:

1. PYTHON BACKGROUND

Python is a programming language that has become quite popular as of 2022. That being said, there's a lot more to it.

- Some languages are compiled. In compiled languages, an application called a compiler checks the code for basic errors, and transforms the code into a format that is easier for a computer to execute. Python is an interpreted script language. This means that the code is intepreted by the computer and executed by the computer at about the same time.

- Python has meaningful whitespace - we use indentation instead of the curly braces used in Java, JavaScript, and other C-style programming languages. This will be important as we start to write code.

- Python is a multi-paradigm language: it's object-oriented, structured, and functional, depending on your purpose and how you use it.
- In comparison to Java and many other languages, Python has very terse syntax.
- Python has lots of built-in capabilities through modules. You'll be able to leverage the language to do a lot of things - and you won't have to write every line of code necessary to accomplish every thing.

In [ ]:

In [ ]:

In [ ]:

History

- 1989: The Python language was started by Guido van Rossum.
- 1994: Python 1.0 was released.
- 2000: Python 2.0 was released.
    - This was a new version of Python, and is when Python started to get more attention.
- 2008: Python 3.0 was released; this was another new version of Python. It was not backwards compatible (*this means that Python 2 code would not necessarily run as expected in Python 3.0 - in fact, it might not run at all*).
    - Python 3.0 had pretty low adoption at this point and lots of programmers continued to use Python 2.
- 2015: Python 2 End of Life was announced. This means that it wouldn't be further developed or supported.
    - In reality, this didn't happen as expected. Many programmers and companies had not yet migrated to Python 3 and lots of new code was still being written in Python 2.
- 2020: Finally Python 2 is gone.

In [ ]:

In [ ]:

2. WHY LEARN PYTHON

- Ease of comprehension:
    - Easy to learn and fun to use.
    - Its syntax, unlike most computer languages, reads like English.
    - Not as stressful to learn as other programming languages.
- Flexibility: Large and robust standard library, High level programming language, no need to recompile the source code
- Used in many industries: MAANG, formerly FAANG, NASA, Spotify, etc
- Earning potential: Python programmers aren't limited. They can work in fields related to:
    - Scientific and Mathematical Computing.
    - Science and Machine Learning
    - Web Development.
    - System automation and administration.
    - Mapping and geography (GIS software), etc...

In [ ]:

In [ ]:

In [ ]:

In [ ]:

3. COMMENTS

- Comments are a way to explain what's going on in your code, or why it's happening. It's a good way to write notes to your future self when you have to read it later; comments might also be useful to other people who might have to read your code.
- Comments are simple in Python, and it's a good idea to add some substantial comments to your code for clarity. In Python, there are 2 types of comments; single line comments and multi-line comments.
- One thing to note: if you run the code blocks below, you'll find that they don't do anything - and that's the point! Python ignores the comments.

`Single line comments are created using the octothorpe (aka the pound or number sign).`

In [1]:
```
# Most comments in Python start w/ an octothorpe (also known as the pound sign).
```

`Multi line comments can use 3 single quotes.`

In [2]:
```
''' Multi-line comments in Python
    can be created with 3 single quotes '''
```

Out[2]:
```
' Multi-line comments in Python \n    can be created with 3 single quotes '
```

`Multi-line comments can also use 3 double quotes.`

In [3]:
```
"""
  Three
  double
  quotes
  is
  also
  ok
"""
```

Out[3]:
```
' \n  Three\n  double\n  quotes\n  is\n  also\n  ok\n'
```

In [ ]:
```

```

In [ ]:
```

```

4. VARIABLES

Variables are containers for storing data values. A variable is created the moment you first assign a value to it.

Example:

- school = "KNUST"
- first_name = "John"
- x=5

In [ ]:

Convention and rules in variables:

- variable name must start with a letter or the underscore character.
  - e.g: my_name = "John"
- Variable name cannot start with a number.
  - e.g: 2my_name = "John"
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
  - e.g: name2 = "John", PI = 3.142, my_name = "Nat"
- All capital letters are for varialbles that can change.
  - e.g: PI = 3.142, STUDENT="Amanda"
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Variable names must be descriptive. e.g: first_name = "John"

In [4]:
```
#declaring a variable
value = 50
Value = 100
```

Variables can be used to perform calculations. We can start writing sequences of lines of code that perform particular operations.

In [5]:
```
x = 5
y= x + 1
y
```

Out[5]:
```
6
```

In [ ]:

In [ ]:

5. DATA TYPES

INT

- **int** which stands for `integer.`

In [6]:
```
a=2
print(a)
```

```
2
```

FLOAT

- **float** represents the `floating point number`. Float is used to represent real numbers and is written with a decimal point dividing the integer and fractional parts

In [7]:
```
b=5.8
print(b)
```

```
5.8
```

In [ ]:

In [ ]:

BOOL

- **bool** which stands for boolean. Boolean can be `True` or `False`. As we stated above, Boolean values are simple. They can only have the values `True` and `False`. We often use Boolean values with comparison operators. Some common comparison operators you might see:
- `==` : equal to, as in `100.0 == 100`
- `!=` : not equal to, as in `35.2 != 550`
- `>` : greater than, as in `7 > 4`
- `<` : less than, as in `3.5 < 3.7`
- `>=` : greater than or equal to, as in `100 >= 99` as well as `100 >= 100`
- `<=` : less than or equal to, as in `333 <= 400` as well as `400 <= 400`

In [8]:

```
print(10 > 9)
print(3 > 5)
print(12 / 2 == 6)
print(12 / 3 != 4)
print(7 <= 3)
```

```
True
False
True
False
False
```

Note that in Python, we use capital letters for `True` and `False` - this is different from many other programming languages.

In [9]:

```
m = True
n = True
p = False
q = False

print(n)
```

```
True
```

You can use `and` as well as `or` to combine Boolean comparison.

- If x **and** y are both true, `x and y` will be `True`
- If either x or y are false, `x and y` will be `False`
- If either x or y are true, or they are both true, `x or y` will be `True`.
- If both x and y are false, `x or y` will be `False`.

Try it below and see for yourself!

In [10]:

```
print(m and p)  # p is False, so this will be False
print(m or p)   # m is True, so this will be True
print(m and n)  # both are True, so this will be True
print(p or q)   # both are False, so this will be False
```

```
False
True
True
False
```

In [ ]:

In [ ]:

```

```

There's also `not`, for negation.

In [11]:

```
print(not q)
print(not n)
print(m or not p)
print(m and not n)
```

```
True
False
True
False
```

All possible `and` combinations are below.

In [12]:

```
print(True and True)
print(False and True)
print(True and False)
print(False and False)
```

```
True
False
False
False
```

All possible `or` combinations are below.

In [13]:

```
print(True or True)
print(False or True)
print(True or False)
print(False or False)
```

```
True
True
True
False
```

STR

- **str** which stands for `string.` When we want to represent text, we use strings. A **string** is just a bunch of characters (letters, numbers, or different symbols) that we associate together.

Strings in Python can be delimited by either single or double quotes.

In [14]:

```
r = 'Single quote string.'
s = "Double quote string."

print(r)
print(s)
```

```
Single quote string.
Double quote string.
```

In Python we use a `len` function to retrieve the number of characters in a string.

In [15]:

```
r = 'abc'
print(len(r))
```

```
3
```

In [ ]:

```

```

In [ ]:

Multi-line comments are actually valid strings in Python, so you'll sometimes see this used when a really long string is needed.

In [16]:
```
multi_line = '''Some really long string.
It can take up multiple lines.
The multi-line comment is really just a string and that's just valid syntax.'''
print(multi_line)
```

```
Some really long string.
It can take up multiple lines.
The multi-line comment is really just a string and that's just valid syntax.
```

You can also concatenate strings using the plus sign (`+`).

In [17]:
```
c = 'cat'
d = 'dog'
f = 'fish'

pet = d

sentence = 'My favorite pet is a ' + pet
print(sentence)
```

```
My favorite pet is a dog
```

Multiple strings can be concatenated together, too.

In [18]:
```
f = 'fox'
b = 'brown'
space = ' '

sentence = 'The' + space + f + space + 'is' + space + b + '.'
print(sentence)
```

```
The fox is brown.
```

String Methods

In [19]:
```
course = "Python Programming"
```

In [20]:
```
# To convert string to lower case
print(course.lower())
```

```
python programming
```

In [21]:
```
# To convert string to upper case
print(course.upper())
```

```
PYTHON PROGRAMMING
```

In [22]:
```
# To capitalize the first letter of each word
print(course.title())
```

```
Python Programming
```

In [23]:
```
# To check the existence of character
print("pro" in course)
```

```
False
```

In [ ]:

In [ ]:

In [ ]:

Python also supports "slices" - you can directly access entire parts of a string. You do this similar to an index, but you use 2 numbers separated by a colon (`:`). The first number is the starting index of the string and the second number is the first index you **don't** want to include.

In [24]:

```
s = 'the quick brown fox jumped over the lazy dog'

print(s[0:3])
print(s[16:19])
```

```
the
fox
```

You can also omit the first or second value in the slice. If the second value is omitted, the rest of the string is included; if the first value is omitted, everything in the string up to the second value is included.

In [25]:

```
s = 'the quick brown fox jumped over the lazy dog'

print(s[:9])
print(s[36:])
```

```
the quick
lazy dog
```

You can even use negative numbers to access values starting from the end of the string.

In [26]:

```
s = 'the quick brown fox jumped over the lazy dog'

print(s[-1])
print(s[-2])
print(s[-3])
```

```
g
o
d
```

Negative numbers work with slices, too.

In [27]:

```
s = 'the quick brown fox jumped over the lazy dog'

# 12 characters from the end, up to but not including the last 4 characters
print(s[-12:-4])

# the last 24 characters in the string
print(s[-24:])

# the beginning of the string, up to but not including the last 18 characters
print(s[:-18])
```

```
the lazy
jumped over the lazy dog
the quick brown fox jumped
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

6. PYTHON OPERATORS

The various types of Python operators are

- Arithmetic,
- Comparison, and
- Logical.

Arithmetic Operators

They are +, -, *, /, //, %, and **

In [28]:

```
print(10+5)
print(10-5)
print(10*5)
print(10/5)
print(10//5)
print(10%5)
print(10**5)
print(10%5+2)
print(10**5*2)
print(2**(5*2))
```

```
15
5
50
2.0
2
0
100000
2
200000
1024
```

Order of Precedence	Operator	Meaning
[1]	**	Exponentiation
[2]	/	Division
[2]	*	Multiplicative
[3]	+	additive
[3]	-	Minus

- Within the same level of precedence, evaluation will proceed from left to right for all the operators with the exception of exponentiation where evaluation proceeds from right to left.
- Parentheses (...) can be used to change the default order of precedence.

In [ ]:

In [ ]:

In [ ]:

Comparison Operators

- Comparison operators are used to test the relationship between two numeric operands for that specific operator.
- Expression involving these operators will reduce to either True or False.

Operator	Meaning
<	Less than
>	Greater than
==	Identically equal to
!=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

Logical Operator

- Logical operators act on logical expressions to create a largest logical expression.
- Parentheses (...) can be used to change the default order of precedence.

Operator	Meaning
and	Returns True if both statements are true
or	Returns True if one of the statements is true
not	Reverse the result, returns False if the result is true

In [ ]:

7. CONDITIONAL STATEMENT

- The `if` statement is a decision-making statement that guides a program to make decisions based on specified criteria. The "if" statement executes one set of code if a specified condition is met (True) or another set of code evaluates to False.
- The `elif` statement allows you to check multiple expressions for True and execute a block of code as soon as one of the conditions evaluates to True.
- The `else` block allows a program to be executed if none of them is true.

N.B Comparison and logical operators are used to specify then condition

In [29]:

```
temperature = 30
if temperature > 30:
    print("Drink water")
else:
    print("Sip tea")
```

```
Sip tea
```

In [ ]:

`Example:` Write a program that calculates the cwa of each student.

In [30]:
```
#solution
cwa = 67
if cwa >= 70.0:
    print("First class")
elif (cwa < 70.0) and (cwa >= 60.0):
    print("Second class upper")
elif (cwa < 60.0) and (cwa >= 50.0):
    print("Second class lower")
elif (cwa < 50.0) and (cwa >= 40.0):
    print("pass")
else:
    print("fail")
```

```
Second class upper
```

In [ ]:

8. LOOPS

- Loops are used to create repetitions.
- Loops are used to iterate through object (like a list, tuple, set, etc.) and perform the same action for each entry.
- There are three types of loops. We have `for loop`, `while loop` and `nested loop`.

For Loop

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Examples of For Loop

In [31]:
```
for i in range(2):
    print(i)
```

```
0
1
```

In [32]:
```
for i in range(1, 2):
    print(i)
```

```
1
```

In [33]:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

```
apple
banana
cherry
```

In [ ]:

In [ ]:

While Loop

- With the while loop we can execute a set of statements as long as a condition is true.

Examples of While Loop

In [34]:
```
i=1
while i < 6:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

In [ ]:

9. FUNCTIONS

A function is a bit of code that performs some operation. Functions allow us to reuse our code efficiently - we can write a function to do something, and then reuse that function over and over again.

Another great feature of functions is that they allow us to replace code with something that is much more readable. For example, the `print()` function just prints - we can kind of intuitively understand what it does without having to understand all of its implementation.

We've already been using a few functions:

- `len()` for the length of a string or a list.
- `type()` for the type of data we're using.
- `print()` for output to the user.
- `input()` for input from the user.

Built-in Functions

Python has some other useful built-in functions. These can just be used as necessary. A few others:

- `abs()` : absolute value of a number.
- `round()` : round a number to a particular number of digits.
- `min()` : retrieve the minimum of a list of values.
- `max()` : retrieve the maximum of a list of values.
- `sorted()` : retrieve a sorted version of a group of values.

Let's try them out!

In [35]:
```
# abs() - This function will return the absolute value of a number - it's pretty straigh
print(abs(12.5))
print(abs(-9))
```

```
12.5
9
```

In [ ]:

In [36]:
```
# round() - Rounds a number to a certain number of decimal places (default is 0).
print(round(12.51))
print(round(-1111.234232, 1))
```

```
13
-1111.2
```

In [37]:
```
# min() - Returns the minimum value from a sequence of values.
# This works on lists, sets, and dictionaries (keys), too.
# You can also just supply a bunch of parameters to the function directly.

list_values = [2, 8, 1, 4, 1, 3, -3, 1, 0]
min_value = min(list_values)
print(min_value)

other_values = [1, 2, 3, 4, 5, -1, -2, -3, -4, -5, 10, 9, 8]
print(min(other_values))
```

```
-3
-5
```

In [38]:
```
# max() - Returns the maximum value from a sequence of values - kind of the opposite of 

list_values = [2, 8, 1, 4, 1, 3, -3, 1, 0]
max_value = max(list_values)
print(max_value)

other_values = [1, 2, 3, 4, 5, -1, -2, -3, -4, -5, 10, 9, 8]
print(max(other_values))
```

```
8
10
```

In [39]:
```
# sorted() - Creates a sorted version of a list. Unlike min() this won't work on a seque
# Note that unlike the .sort() method, it doesn't affect the original list.
# The original list is still unsorted.
values = [1, 2, 3, 4, 5, -1, -2, -3, -4, -5, 10, 9, 8]
print(sorted(values))
print('The original is not sorted!')
print(values)
```

```
[-5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 8, 9, 10]
The original is not sorted!
[1, 2, 3, 4, 5, -1, -2, -3, -4, -5, 10, 9, 8]
```

In any programming language, code reuse is going to be an important, time saving feature. One of the most basic and straightforward ways we can reuse our code is by writing our `own functions`. Think of functions as tiny, self-contained programs that perform some specific action or give us some results.

Python functions are declared with the `def` keyword. We give our function a name (in this case `say_hello`) and any parameters that it might use (in this case, there are no parameters).

In [40]:
```
def say_hello():
    print('Hello!')
```

For this to work, it has to be `invoked` or `called`.

In [41]:
```
say_hello()
```

```
Hello!
```

In [ ]:

In [42]:
```
#defining a function called identity
def identity(first_name, last_name):
    return "hi, my firstname is " + first_name + " and my lastname " + last_name


print(identity("Thomas", "Kyeimiah"))
```

```
hi, my firstname is Thomas and my lastname Kyeimiah
```

In [ ]:

10. DATA STRUCTURES

- The basic Python data structures in Python include list, set, tuples, and dictionary. Each of the data structures is unique in its own way.
- Data structures are “containers” that organize and group data according to type.
- The data structures differ based on mutability and order

List

- Lists are used to store multiple items in a single variable.
- Lists are created using square brackets, thus `[ ]`.

In [43]:
```
#example
a= ["Thomas", 4, "Nathaniel"]
a
```

Out[43]:
```
['Thomas', 4, 'Nathaniel']
```

To get the length in a list the `len()` is used.

In [44]:
```
print(len(a))
```

```
3
```

To access `item` in a list

In [45]:
```
print(a[0])
print(a[:2])
print(a[::-1])
```

```
Thomas
['Thomas', 4]
['Nathaniel', 4, 'Thomas']
```

To `loop` through list.

In [46]:
```
a= ["Thomas", 4, "Nathaniel"]
for i in a:
    print(i)
```

```
Thomas
4
Nathaniel
```

In [ ]:

To add `new element` at the end of the list.

In [47]:
```
a.append("TA")
a
```

Out[47]:
```
['Thomas', 4, 'Nathaniel', 'TA']
```

To add `new element` to the `beginning` of the list.

In [48]:
```
a.insert(0, "Amo")
a
```

Out[48]:
```
['Amo', 'Thomas', 4, 'Nathaniel', 'TA']
```

To `remove` element from the list.

In [49]:
```
a.pop()
a
```

Out[49]:
```
['Amo', 'Thomas', 4, 'Nathaniel']
```

Tuple

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets, thus `( )`.

You can't delete or add element in a tuple, but you can access an element like a list

In [50]:
```
b= ("Thomas", 4, "Nathaniel")
b
```

Out[50]:
```
('Thomas', 4, 'Nathaniel')
```

In [51]:
```
print(b[0])
print(b[:2])
```

```
Thomas
('Thomas', 4)
```

Set

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed.
- A set are written with curly brackets, thus `{ }`
- Duplicates are not allowed in sets.

In [52]:
```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

```
{'cherry', 'apple', 'banana'}
```

Set shine in the mathematical world. It embraces the use of union, intersection set, etc...

In [53]:
```
a = {1,1,2,3,4}
b = {1, 5}
```

In [ ]:

In [54]:
```
#To find the union of a and b
print(a | b)
```

```
{1, 2, 3, 4, 5}
```

In [55]:
```
#To find the intersection of a and b
print(a & b)
```

```
{1}
```

In [56]:
```
#To find the difference between a and b meaning elements that are in a but not in b
print(a - b)
```

```
{2, 3, 4}
```

Dictionary

- Dictionaries are used to store data values in `key:value` pairs.
- Dictionaries are written with `curly brackets`, and have `keys` and `values`

In [57]:
```
#example
brand_type = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(brand_type)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

To access a `value` of a key, we use `square bracket`

In [58]:
```
#finding the value of the key 'brand'
brand_type = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(brand_type["brand"])
```

```
Ford
```

To loop through dictionary

In [59]:
```
for key, value in brand_type.items():
    print(key, value)
```

```
brand Ford
model Mustang
year 1964
```

In [ ]:

In [ ]:

In [ ]:

11. MODULES

One of the benefits of using Python is its large library of capable, built-in, easy-to-use `modules`. Python has libraries for almost every activity. With advancing technologies, libraries are almost widely and readily available and can be called into the Python terminal for use, with the import function. Some of these libraries include pandas, numpy, matplotlib, xarray, scipy, etc. In this course, our emphasis will be on `numpy`, `matplotlib`, `pandas`, `xarray`, to name a few.

IMPORTING/LOADING PACKAGES

- **Option 1:** import package
- **Option 2:** import package as pkg
- **Option 3:** from package import pkg_item
- **Option 4:** from package import pkg_item as pkg

In [ ]:

```

```

In [ ]:

```

```

12. PANDAS

Pandas, which stands for `Python Data Analysis Library` is an open source Python package that is most widely used for data science/data analysis and machine learning tasks. It is built on top of another package named Numpy, which provides support for multi-dimensional arrays. Pandas is used to explore, analyse, manipulate data, etc

WHY PANDAS

- Simple to Use
- Integrated with many other data science and machine learning tools
- Helps to get your data ready for machine learning and analysis

Importing Pandas

`import pandas`

`import pandas as pd` (recommended)

PANDA DATA STRUCTURE

There are two main data structures in pandas `Series` and `DataFrame`

`Series` is one-dimensional array which takes in a list as an arguement. `DataFrame` is two-dimensional array takes in a dictionary as an arguement.

In [60]:

```
import pandas as pd
```

In [61]:

```
#example of Series
data = ['BMW', 'Toyota', 'Honda']
cars = pd.Series(data)
cars
```

Out[61]:

```
0       BMW
1    Toyota
2     Honda
dtype: object
```

In [62]:

```
data = {
    'name': ['Daniel', 'Oppong', 'Asamoah'],
    'school': ['KNUST', 'Legon', 'UCC'],
    'program': ['Meteorology', 'Biochemistry', 'Sociology'],
    'level': [200, 400, 100]
}
```

In [63]:
```
details = pd.DataFrame(data)
details
```

Out[63]:

	name	school	program	level
0	Daniel	KNUST	Meteorology	200
1	Oppong	Legon	Biochemistry	400
2	Asamoah	UCC	Sociology	100

IMPORT EXISTING DATA FILES

There are various files that we can work. We have **text files**, **csv files**, **netcdf files**, etc..., For now we will be focusing on working with .csv and .nc files

In [64]:
```
# to read csv files
car_sales = pd.read_csv('car-sales.csv')
car_sales
```

Out[64]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

In [ ]:

DESCRIBING DATAFRAME

In [65]:
```
# attributes
car_sales.dtypes
```

Out[65]:
```
Make             object
Colour           object
Odometer (KM)     int64
Doors             int64
Price            object
dtype: object
```

In [66]:
```
# to get the column of a DataFrame
car_sales.columns
```

Out[66]:
```
Index(['Make', 'Colour', 'Odometer (KM)', 'Doors', 'Price'], dtype='object')
```

In [ ]:

In [ ]:

In [ ]:

In [67]:

```
# to get the statistical information about numerical column
car_sales.describe()
```

Out[67]:

	Odometer (KM)	Doors
count	10.000000	10.000000
mean	78601.400000	4.000000
std	61983.471735	0.471405
min	11179.000000	3.000000
25%	35836.250000	4.000000
50%	57369.000000	4.000000
75%	96384.500000	4.000000
max	213095.000000	5.000000

In [68]:

```
# to get information about a DataFrame
car_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   Make           10 non-null     object
 1   Colour         10 non-null     object
 2   Odometer (KM)  10 non-null     int64 
 3   Doors          10 non-null     int64 
 4   Price          10 non-null     object
dtypes: int64(2), object(3)
memory usage: 528.0+ bytes
```

In [69]:

```
# to get the sum
car_sales.sum()
```

Out[69]:

```
Make                 ToyotaHondaToyotaBMWNissanToyotaHondaHondaToyo...
Colour                   WhiteRedBlueBlackWhiteGreenBlueBlueWhiteWhite
Odometer (KM)                                                   786014
Doors                                                               40
Price                $4,000.00$5,000.00$7,000.00$22,000.00$3,500.00...
dtype: object
```

SELECTING AND VIEWING DATA WITH PANDAS

In [70]:

```
#viewing the first five rows of a datafame
car_sales.head()
```

Out[70]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00

In [ ]:

In [ ]:

In [71]:
```
#viewing the first seven rows of a datafame
car_sales.head(7)
```

Out[71]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00

In [72]:
```
#viewing the last five rows of a datafame
car_sales.tail()
```

Out[72]:

	Make	Colour	Odometer (KM)	Doors	Price
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

In [73]:
```
#viewing the last seven rows of a datafame
car_sales.tail(7)
```

Out[73]:

	Make	Colour	Odometer (KM)	Doors	Price
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

loc and iloc

loc is used to subset(slice) **index** of a dataframe/series whiles and iloc is used to subset(slice) the **position** of a dataframe/series

In [74]:
```
animals = pd.Series(['cat', 'dogs', 'goat', 'pig'],index=[0, 2, 4, 6])
animals
```

Out[74]:
```
0     cat
2    dogs
4    goat
6     pig
dtype: object
```

In [ ]:

In [75]:
```
#loc --takes the position
animals.loc[4]
```
Out[75]:
```
'goat'
```

In [76]:
```
# iloc -- takes the index
animals.iloc[3]
```
Out[76]:
```
'pig'
```

In [77]:
```
#iloc example on dataframe
car_sales
```
Out[77]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

In [78]:
```
#selecting the first five rows
car_sales.iloc[0:5]
```
Out[78]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00

In [79]:
```
#selecting all the rows and the first column
car_sales.iloc[:,0]
```
Out[79]:
```
0    Toyota
1     Honda
2    Toyota
3       BMW
4    Nissan
5    Toyota
6     Honda
7     Honda
8    Toyota
9    Nissan
Name: Make, dtype: object
```

In [ ]:

In [ ]:

Selecting Columns

Columns can be selected using the **dot notation** and **square bracket**

In [80]:
```
# Using the dot notation
car_sales.Colour
```

Out[80]:
```
0    White
1      Red
2     Blue
3    Black
4    White
5    Green
6     Blue
7     Blue
8    White
9    White
Name: Colour, dtype: object
```

In [81]:
```
# Using square bracket
car_sales['Colour']
```

Out[81]:
```
0    White
1      Red
2     Blue
3    Black
4    White
5    Green
6     Blue
7     Blue
8    White
9    White
Name: Colour, dtype: object
```

In [82]:
```
# to add new column --- use the square bracket
car_sales['New'] = 6
car_sales
```

Out[82]:

	Make	Colour	Odometer (KM)	Doors	Price	New
0	Toyota	White	150043	4	$4,000.00	6
1	Honda	Red	87899	4	$5,000.00	6
2	Toyota	Blue	32549	3	$7,000.00	6
3	BMW	Black	11179	5	$22,000.00	6
4	Nissan	White	213095	4	$3,500.00	6
5	Toyota	Green	99213	4	$4,500.00	6
6	Honda	Blue	45698	4	$7,500.00	6
7	Honda	Blue	54738	4	$7,000.00	6
8	Toyota	White	60000	4	$6,250.00	6
9	Nissan	White	31600	4	$9,700.00	6

In [ ]:

In [ ]:

FILLNA, DROPNA, DROP

- ***fillna*** is use to fill NaN values in a dataframe
- ***dropna*** is use to remove NaN values in a dataframe
- ***drop*** is use to remove a specific column or row

In [83]:
```
missing_data = pd.read_csv('car-sales-missing.csv')
missing_data
```

Out[83]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
2	Toyota	Blue	NaN	3.0	$7,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500
5	Toyota	Green	NaN	4.0	$4,500
6	Honda	NaN	NaN	4.0	$7,500
7	Honda	Blue	NaN	4.0	NaN
8	Toyota	White	60000.0	NaN	NaN
9	NaN	White	31600.0	4.0	$9,700

In [84]:
```
# to fill NaN values with a value
missing_data.fillna(4)
```

Out[84]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
2	Toyota	Blue	4.0	3.0	$7,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500
5	Toyota	Green	4.0	4.0	$4,500
6	Honda	4	4.0	4.0	$7,500
7	Honda	Blue	4.0	4.0	4
8	Toyota	White	60000.0	4.0	4
9	4	White	31600.0	4.0	$9,700

In [85]:
```
# to remove NaN values from a dataframe
missing_data.dropna()
```

Out[85]:

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500

In [ ]:

In [86]:
```
# to remove a specific column from a dataframe
missing_data.drop(columns='Price', axis=1)
```

Out[86]:

	Make	Colour	Odometer	Doors
0	Toyota	White	150043.0	4.0
1	Honda	Red	87899.0	4.0
2	Toyota	Blue	NaN	3.0
3	BMW	Black	11179.0	5.0
4	Nissan	White	213095.0	4.0
5	Toyota	Green	NaN	4.0
6	Honda	NaN	NaN	4.0
7	Honda	Blue	NaN	4.0
8	Toyota	White	60000.0	NaN
9	NaN	White	31600.0	4.0

In [ ]:

13. NUMPY

Numpy basically refers to **NUMerical PYthon** is similar to python list. and is used for numeric actions and calls, ranging from basic to complex numerical functions. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, basic linear algebra, basic statistical operations, random simulation and many more.

WHY NUMPY

- Since it is written in C language, it is faster as compared to python list
- Machine only understands 0's and 1's so using numpy makes the machine understand.

Importing Numpy

`import numpy`

`import numpy as np` (recommended)

In [87]:
```
import numpy as np
```

NUMPY DATATYPES AND ATTRIBUTES

Numpy's main datatype is **ndarray** which stands for n dimensional array

In [88]:
```
n = np.array([[1,2,3],[4,5,6]])
n
```

Out[88]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [ ]:

In [89]:
```
#to know the type of numpy
type(n)
```
Out[89]:
```
numpy.ndarray
```

In [90]:
```
# to check the shape
n.shape
```
Out[90]:
```
(2, 3)
```

In [91]:
```
# to check the size
n.size
```
Out[91]:
```
6
```

In [92]:
```
# to check the number of dimensions
n.ndim
```
Out[92]:
```
2
```

In [93]:
```
# to check the datatype
n.dtype
```
Out[93]:
```
dtype('int64')
```

Manipulating Numpy Arrays

In [94]:
```
a1 = np.array([1,2,3])
a2 = np.array([2,5,6])
```

In [95]:
```
#Addition

a1+a2
```
Out[95]:
```
array([3, 7, 9])
```

In [96]:
```
# other way
np.add(a1, a2)
```
Out[96]:
```
array([3, 7, 9])
```

In [97]:
```
# subtraction
a1 - a2
```
Out[97]:
```
array([-1, -3, -3])
```

In [98]:
```
# other way
np.subtract(a1, a2)
```
Out[98]:
```
array([-1, -3, -3])
```

In [99]:
```
# multiplication
a1 * a2
```
Out[99]:
```
array([ 2, 10, 18])
```

In [ ]:

In [100…
```
#other way
np.multiply(a1,a2)
```
Out[100]:
```
array([ 2, 10, 18])
```

In [101…
```
# division
a1 / a2
```
Out[101]:
```
array([0.5, 0.4, 0.5])
```

In [102…
```
#other way
np.divide(a1, a2)
```
Out[102]:
```
array([0.5, 0.4, 0.5])
```

In [103…
```
# floor division
a1 // a2
```
Out[103]:
```
array([0, 0, 0])
```

In [104…
```
# other way
np.floor_divide(a1, a2)
```
Out[104]:
```
array([0, 0, 0])
```

In [105…
```
#exponential
a1**a2
```
Out[105]:
```
array([  1,  32, 729])
```

In [106…
```
#other way
np.power(a1,a2)
```
Out[106]:
```
array([  1,  32, 729])
```

Aggregation

In [107…
```
a3 = np.array([[1,2,3],
             [4,5,6]])
a3
```
Out[107]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [108…
```
# to find the sum
np.sum(a3)
```
Out[108]:
```
21
```

In [109…
```
# to find the mean
np.mean(a3)
```
Out[109]:
```
3.5
```

In [110…
```
# to find the maximum number
np.max(a3)
```
Out[110]:
```
6
```

In [ ]:

In [111…
```
# to find the minimum number
np.min(a3)
```
Out[111]:
```
1
```
In [112…
```
# to find the standard deviation
np.std(a3)
```
Out[112]:
```
1.707825127659933
```
In [113…
```
# to find the variance
np.var(a3)
```
Out[113]:
```
2.9166666666666665
```
In [114…
```
# to find the square root
np.sqrt(a3)
```
Out[114]:
```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```
RESHAPE AND TRANSPOSE

The difference betwen transpose and **reshape** is that **transpose** flip the axis and reshape helps you to create your own custom shape.

In [115…
```
a1 = np.array([[1,2,3],
               [4,5,6]])
a1
```
Out[115]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```
In [116…
```
a1.shape
```
Out[116]:
```
(2, 3)
```
In [117…
```
# reshape
a1_reshaped = a1.reshape(6,1)
a1_reshaped
```
Out[117]:
```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```
In [118…
```
a1_reshaped.shape
```
Out[118]:
```
(6, 1)
```
In [119…
```
# transpose
a_transpose = a1.T
a_transpose
```
Out[119]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [ ]:

In [120…
```
a_transpose.shape
```
Out[120]:
```
(3, 2)
```

Sorting Arrays

In [121…
```
array = np.array([[7, 8, 1, 5, 9],
                  [8, 9, 7, 3, 0],
                  [3, 5, 0, 2, 3]])
array
```
Out[121]:
```
array([[7, 8, 1, 5, 9],
       [8, 9, 7, 3, 0],
       [3, 5, 0, 2, 3]])
```

In [122…
```
# sort array based on values
np.sort(array)
```
Out[122]:
```
array([[1, 5, 7, 8, 9],
       [0, 3, 7, 8, 9],
       [0, 2, 3, 3, 5]])
```

In [ ]:

14. MATPLOTLIB

It is a plotting library in python which allows us to turn our data into visuals

WHY MATPLOTLIB

- Built on top of numpy arrays and python
- Integrate directly with pandas
- Can create basic or advanced plots
- Simple to use interface(once you set the formulations)

Importing Numpy

`import matplotlib`

`import matplotlib.pyplot as plt` (recommended)

`from matplotlib import pyplot as plt` (recommended)

In [123…
```
import matplotlib.pyplot as plt
```

There are two ways of plotting in matplotlib. We have:

- **Pyplot API method**
- **Object Oriented Method**

THe Pyplot API is the default way of plotting. For advanced ploting and customizing, the Object Oriented Method is best.

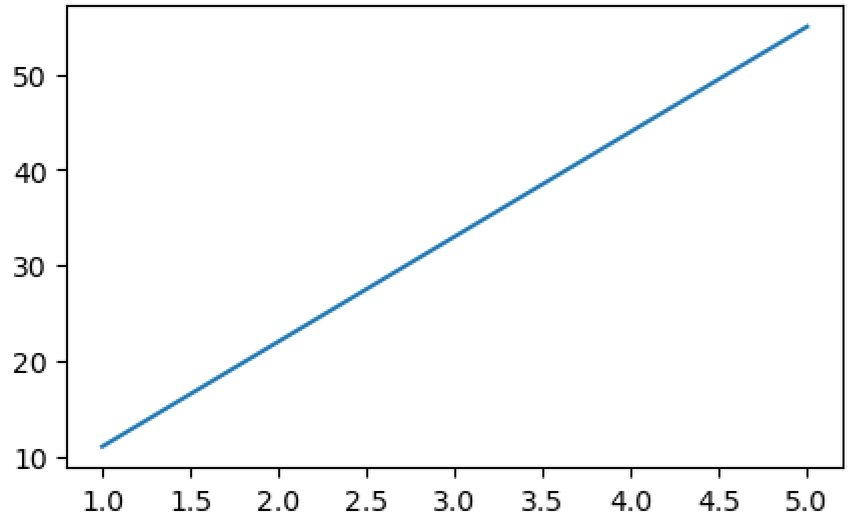
In [ ]:

In [ ]:

PYPLOT API METHOD

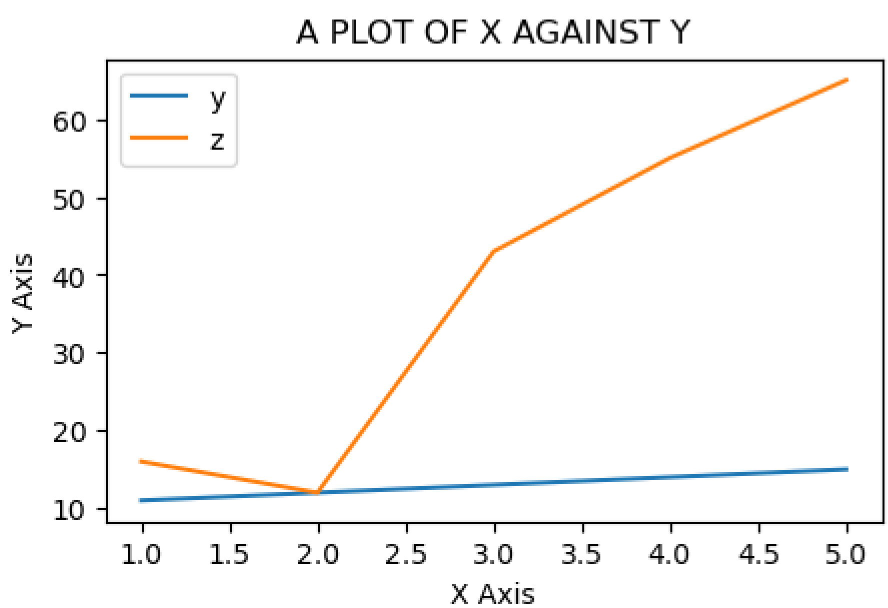
In [124…

```
# example
plt.figure(figsize=(5,3))
x = [1, 2, 3, 4, 5]
y = [11, 22, 33, 44, 55]
plt.plot(x, y)
plt.show()
```



In [125…

```
plt.figure(figsize=(5,3))
x = [1,2,3,4,5]
y = [11,12,13,14,15]
z = [16, 12,43,55,65]
plt.plot(x, y, label='y')
plt.plot(x, z,  label='z')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('A PLOT OF X AGAINST Y')
plt.legend()
plt.show()
```

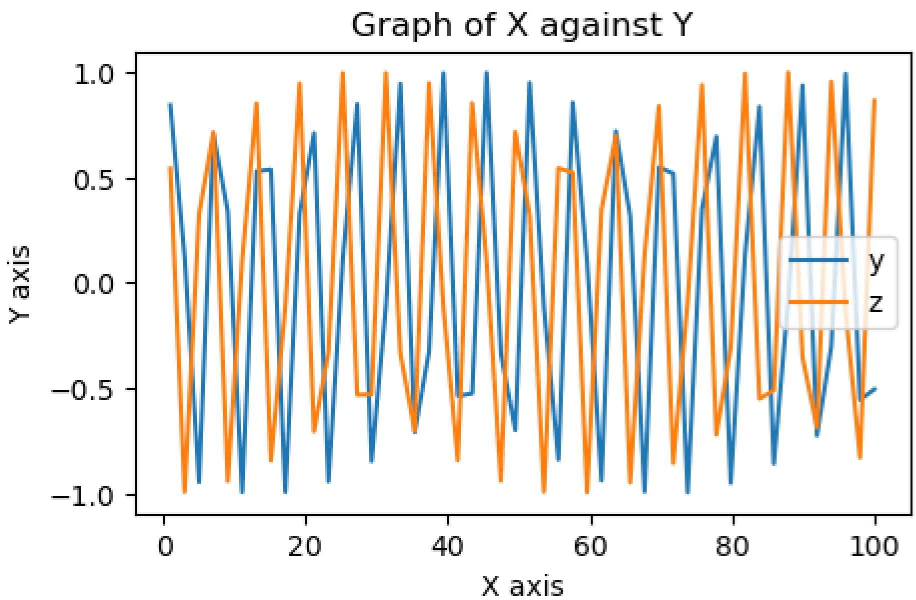


In [ ]:

In [ ]:

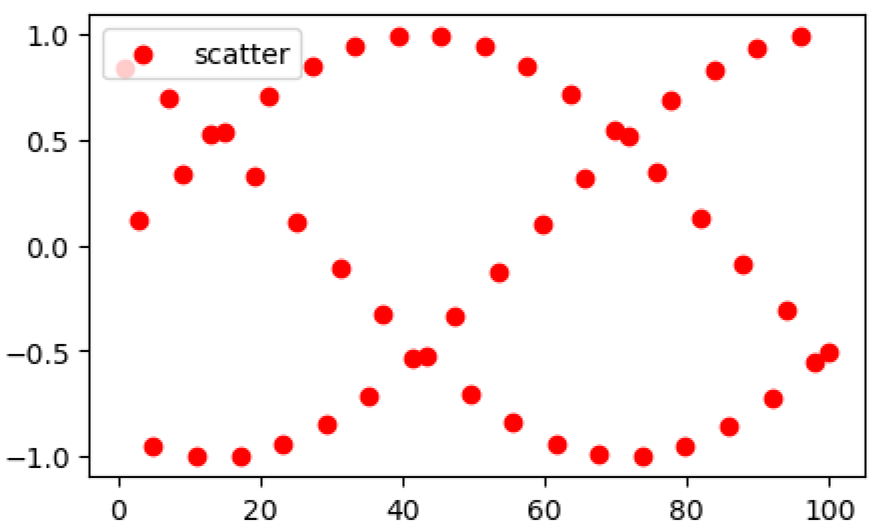
In [126…

```
# line plot
plt.figure(figsize=(5,3))
x = np.linspace(1, 100)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, label='y')
plt.plot(x, z, label='z')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.title('Graph of X against Y')
plt.legend();
```



In [127…

```
#scatter plot
plt.figure(figsize=(5,3))
x = np.linspace(1, 100)
y = np.sin(x)
plt.scatter(x=x, y=y, color='red', label='scatter')
plt.legend();
```



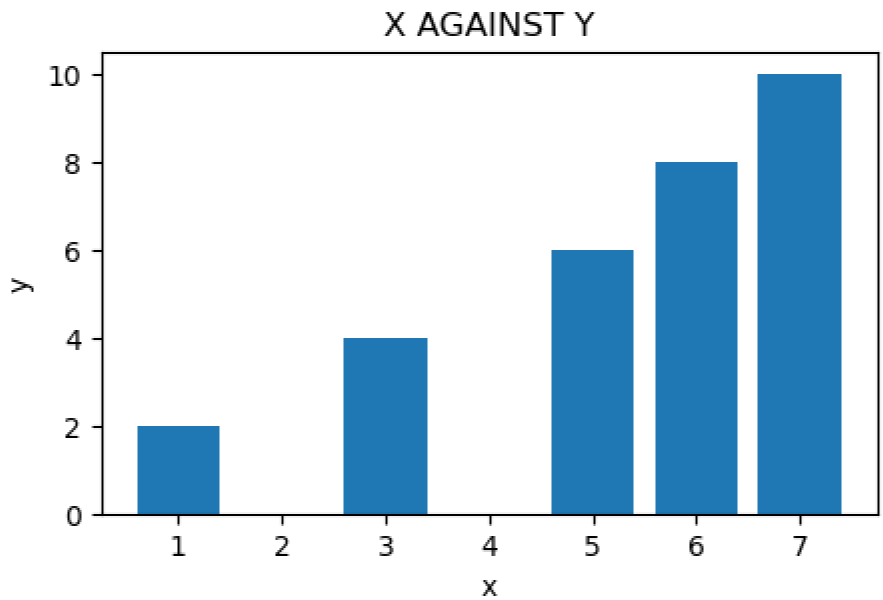
In [ ]:

In [ ]:

In [ ]:

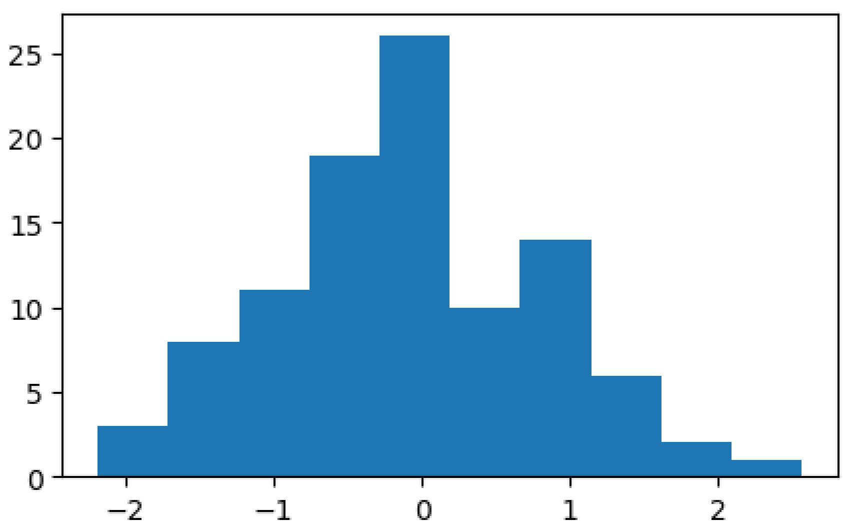
In [128…

```
#bar plot
plt.figure(figsize=(5,3))
x = [1,3,5,6,7]
y = [2,4,6,8,10]
plt.xlabel('x')
plt.ylabel('y')
plt.title('X AGAINST Y')
plt.bar(x,y);
```



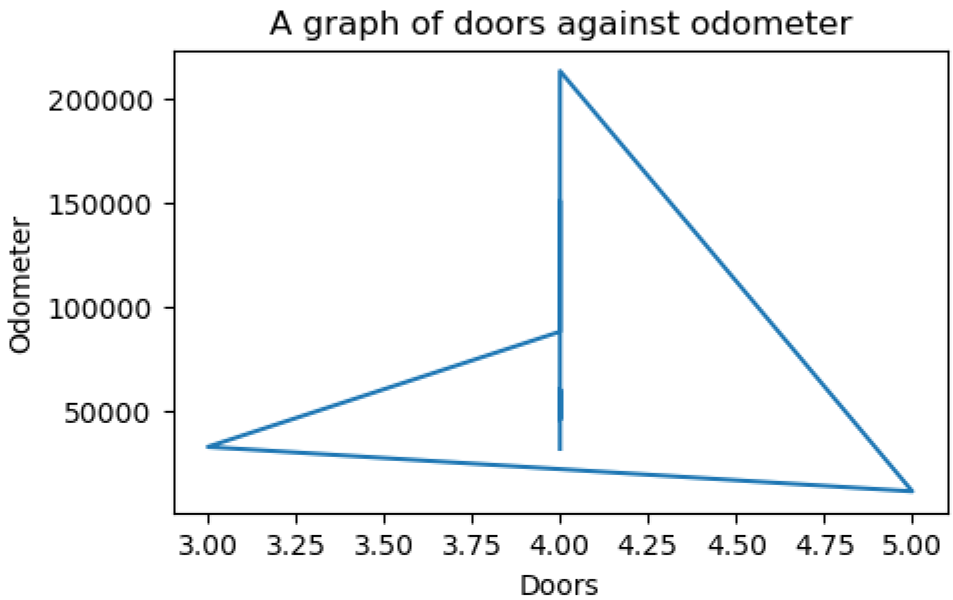
In [129…

```
# Histogram
plt.figure(figsize=(5,3))
x = np.random.randn(100)
plt.hist(x)
plt.show()
```



In [130…

```
# preparing our data and plotting graph
plt.figure(figsize=(5,3))
odometer = car_sales['Odometer (KM)']
doors = car_sales['Doors']
plt.plot(doors, odometer)
plt.xlabel('Doors')
plt.ylabel('Odometer')
plt.title('A graph of doors against odometer');
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

Object Oriented Method

In [131…
```
# ANALYSIS HEART DIESEASE
plt.figure(figsize=(5,3))
#preparing data
heart_disease = pd.read_csv('heart-disease.csv')
over_50 = heart_disease[heart_disease['age'] > 50]
over_50
```

Out[131]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
5	57	1	0	140	192	0	1	148	0	0.4	1	0	1	1
6	56	0	1	140	294	0	0	153	0	1.3	1	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
297	59	1	0	164	176	1	0	90	0	1.0	1	2	1	0
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

208 rows × 14 columns

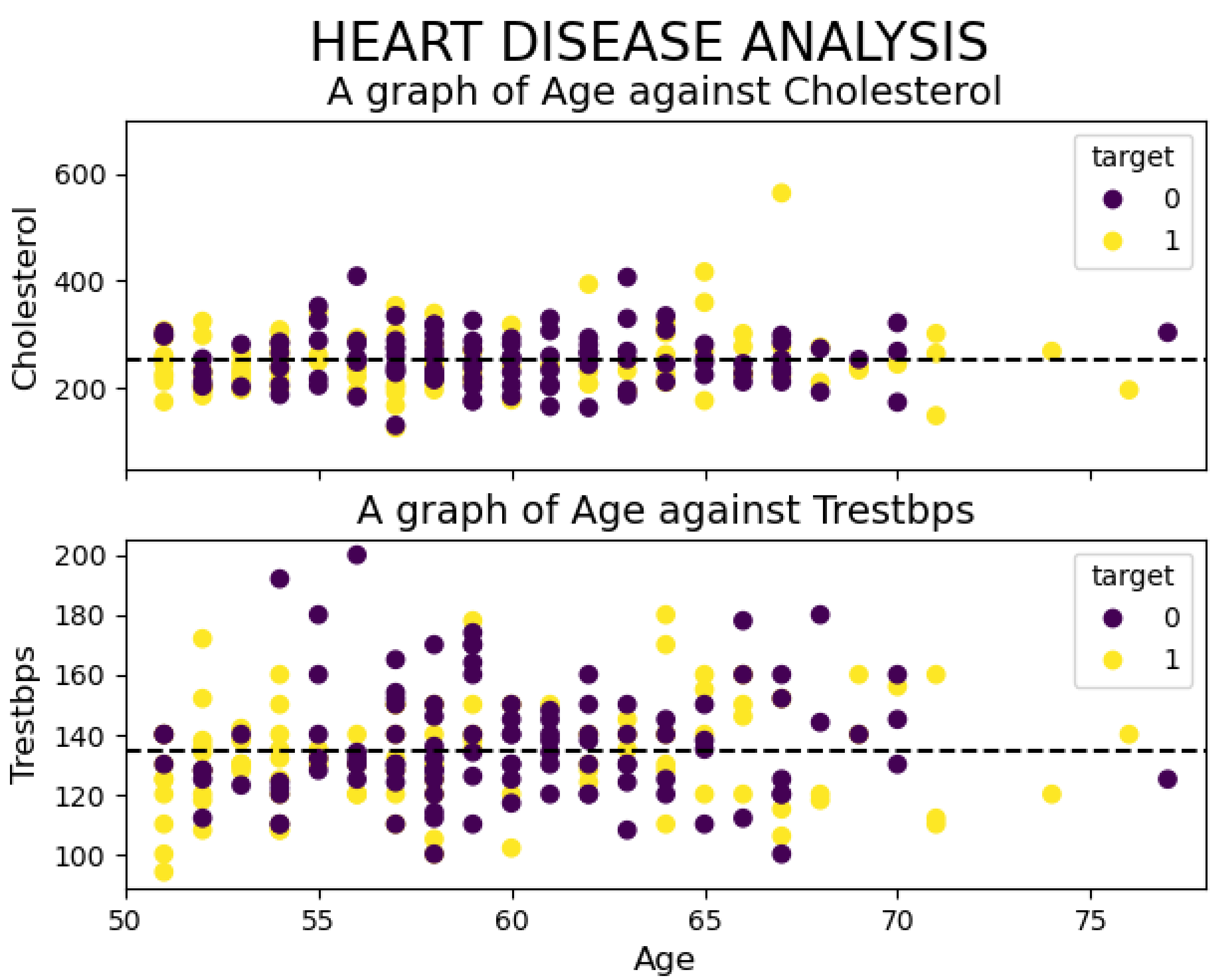
```
<Figure size 500x300 with 0 Axes>
```

In [ ]:

In [ ]:

In [132…

```
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(7,5), sharex=True)
scatter = ax1.scatter(x=over_50['age'],
            y=over_50['chol'],
            c=over_50['target'])

ax1.set_ylabel('Cholesterol', fontsize=12)
ax1.set_title('A graph of Age against Cholesterol', fontsize=14)
ax1.set_xlim([50,78])
ax1.set_ylim([50, 700])
ax1.axhline(y=over_50['chol'].mean(), color='black',linestyle='--')
ax1.legend(*scatter.legend_elements(), title='target')



scatter = ax2.scatter(x=over_50['age'],
            y=over_50['trestbps'],
            c=over_50['target'])
ax2.set_xlabel('Age', fontsize=12)
ax2.set_ylabel('Trestbps', fontsize=12)
ax2.set_title('A graph of Age against Trestbps', fontsize=14)
ax2.set_xlim([50,78])
ax2.axhline(y=over_50['trestbps'].mean(), color='black',linestyle='--')
ax2.legend(*scatter.legend_elements(), title='target')


fig.suptitle('HEART DISEASE ANALYSIS', fontsize=19)
plt.show()
#plt.savefig('hd.png', dpi=100)
```

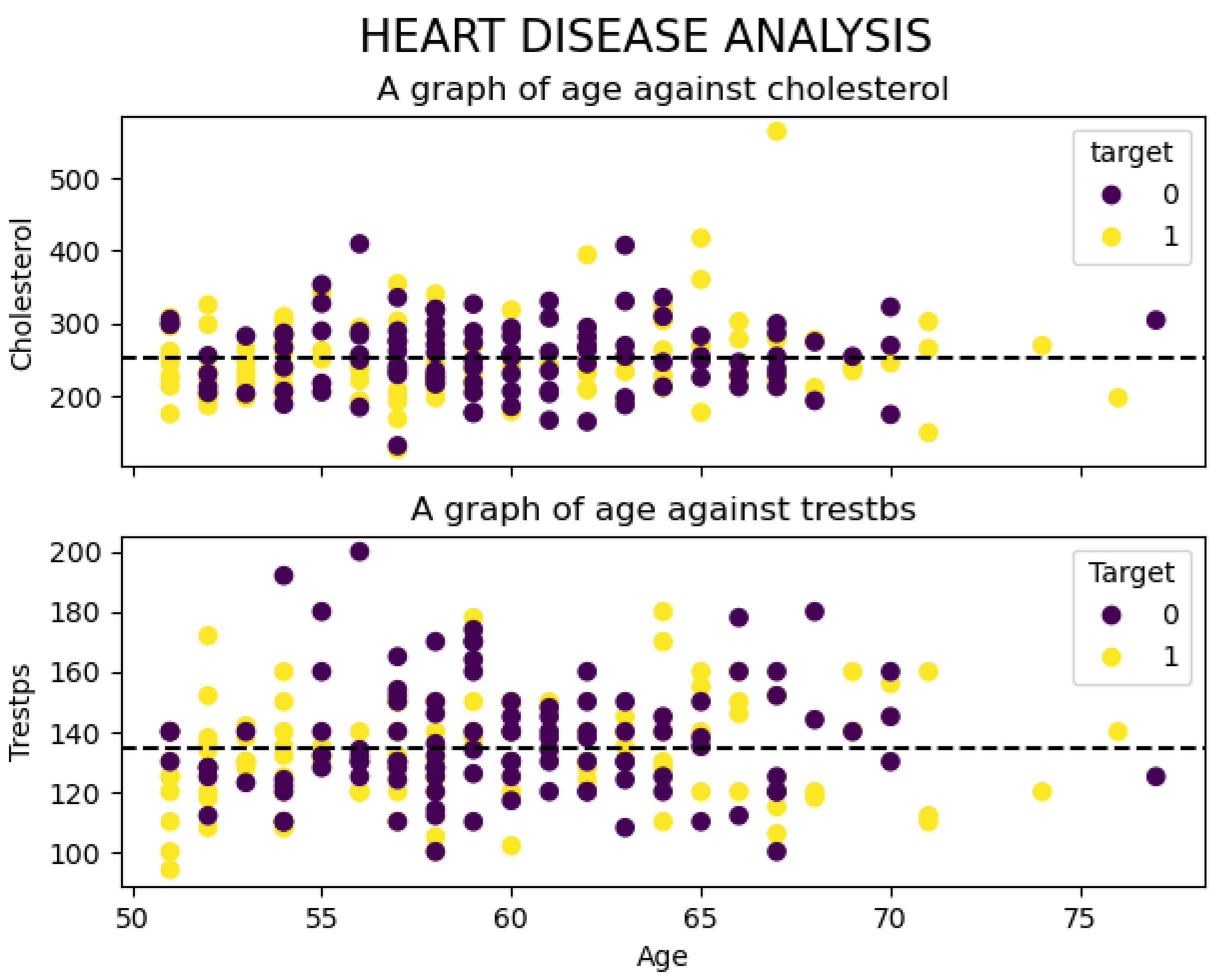


In [ ]:

In [ ]:

In [133…

```
#alternatively
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True, figsize=(7, 5))
ax = ax.flatten()
target=over_50['target']
for i, v in enumerate(ax):
    if i == 0:
        scatter=ax[i].scatter(x=over_50['age'], y=over_50['chol'], c=target)
        #ax[i].set_xlabel('Age')
        ax[i].set_ylabel('Cholesterol')
        ax[i].set_title('A graph of age against cholesterol')
        ax[i].axhline(y=over_50['chol'].mean(), color='black', linestyle='--')
        legend1=ax[i].legend(*scatter.legend_elements(),loc="upper right",
                             title="target")
    if i == 1:
        scatter=ax[i].scatter(x=over_50['age'], y=over_50['trestbps'], c=target)
        ax[i].set_xlabel('Age')
        ax[i].set_ylabel('Trestps')
        ax[i].set_title('A graph of age against trestbs')
        ax[i].axhline(y=over_50['trestbps'].mean(), color='black', linestyle='--')
        legend2=ax[i].legend(*scatter.legend_elements(),loc="upper right",
                             title="Target")
fig.suptitle('HEART DISEASE ANALYSIS', fontsize=16, color='black');
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

15. Introduction to xarray

- Unlabeled, N-dimensional arrays of numbers (e.g., NumPy’s ndarray) are the most widely used data structure in scientific computing. However, they lack a meaningful representation of the metadata associated with their data. Implementing such functionality is left to individual users and domain-specific packages.
- xarry expands on the capabilities of NumPy arrays, providing a lot of streamline data manipulation.
- Xarray's interface is based largely on the netCDF data model (variables, attributes, and dimensions), but it goes beyond the traditional netCDF interfaces to provide functionality similar to netCDF-java's Common Data Model (CDM).
- xarray is motivated by weather and climate use cases.

Core Data Structures

- xarray has 2 fundamental data structures:
    - `DataArray` , which holds single multi-dimensional variables and its coordinates
    - `Dataset` , which holds multiple variables that potentially share the same coordinates

Opening a dataset

An xarray dataset is a container for data and it's associated metadata, including labelled coordinates.

First step, import the xarray package

In [134…

```
import xarray as xr
```

When opening a netCDF file, the file metadata is read and stored as an `xarray.DataSet` .

In [135…

```
ds = xr.open_dataset('SOIL_MOISTURE.nc')
```

The metadata for this dataset is now stored in the variable `ds` . A more informative name could be chosen, but `ds` is fast to type! To examine the contents it is sufficient to just put the variable name in a cell and evaluate it, which is equivalent to `print(ds)` in a python program

In [ ]:

In [ ]:

In [136…
```
ds
```

Out[136]: xarray.Dataset

▸ Dimensions: (**longitude**: 33, **latitude**: 37, **time**: 504)

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▾ Data variables:

stl1	(time, latitude, longitude)	float32	...
stl2	(time, latitude, longitude)	float32	...
stl3	(time, latitude, longitude)	float32	...
stl4	(time, latitude, longitude)	float32	...
swvl1	(time, latitude, longitude)	float32	...
swvl2	(time, latitude, longitude)	float32	...
swvl3	(time, latitude, longitude)	float32	...
swvl4	(time, latitude, longitude)	float32	...

▾ Attributes:

Conventions : CF-1.6

history : 2022-07-08 10:10:30 GMT by grib_to_netcdf-2.25.1: /opt/ecmwf/mars-client/bin/grib_to_netcdf.bin -S param -o /cache/data9/adaptor.mars.internal-1657275023.2669723-2621-7-23d9a0bf-7bf2-4764-8a86-5d53fd492b90.nc /cache/tmp/23d9a0bf-7bf2-4764-8a86-5d53fd492b90-adaptor.mars.internal-1657274477.0104525-2621-5-tmp.grib

`Dataset`

- Xarray's `Dataset` is a dict-like container of labeled arrays (`DataArrays`) with aligned dimensions. It is designed as an in-memory representation of a netCDF dataset. In addition to the dict-like interface of the dataset itself, which can be used to access any `DataArray` in a `Dataset` . Datasets have the following key properties:

Attribute	Description
`data_vars`	OrderedDict of `DataArray` objects corresponding to data variables.
`dims`	dictionary mapping from dimension names to the fixed length of each dimension (e.g., { `lat` : 6, `lon` : 6, `time` : 8}).
`coords`	a dict-like container of arrays (coordinates) that label each point (e.g., 1-dimensional arrays of numbers, datetime objects or strings)
`attrs`	OrderedDict to hold arbitrary metadata pertaining to the dataset.

Accessing the data

The `open_dataset` command only reads the metadata from the netCDF file. It does not attempt to read any data until there is an operation that requires this.

In [ ]:

The `xarray.DataSet` object has a number of methods for accessing the coordinates, attributes and data. The data variables are saved in a `dict`-like structure, `ds.data_vars`:

In [137…

```
ds.data_vars
```

Out[137]:

```
Data variables:
    stl1     (time, latitude, longitude) float32 ...
    stl2     (time, latitude, longitude) float32 ...
    stl3     (time, latitude, longitude) float32 ...
    stl4     (time, latitude, longitude) float32 ...
    swvl1    (time, latitude, longitude) float32 ...
    swvl2    (time, latitude, longitude) float32 ...
    swvl3    (time, latitude, longitude) float32 ...
    swvl4    (time, latitude, longitude) float32 ...
```

An individual variable can be accessed using it's name, either as a `dict` like key

In [138…

```
ds['stl1']
```

Out[138]: xarray.DataArray 'stl1' (**time**: 504, **latitude**: 37, **longitude**: 33)

```
[615384 values with dtype=float32]
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▼ Attributes:

units :	K
long_name :	Soil temperature level 1
standard_name :	surface_temperature

For ease of use xarray also provides access to data variables as a python attribute

In [139…

```
stl1=ds.stl1
stl1
```

Out[139]: xarray.DataArray 'stl1' (**time**: 504, **latitude**: 37, **longitude**: 33)

```
[615384 values with dtype=float32]
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▼ Attributes:

units :	K
long_name :	Soil temperature level 1
standard_name :	surface_temperature

In [ ]:

In [ ]:

So `ds.stl1` is an `xarray.DataArray` and has it's own metadata giving more information about the variable itself. In this case it is near-surface air temperature in Kelvin.

`DataArray`

The DataArray is xarray's implementation of a labeled, multi-dimensional array. It has several key properties:

Attribute	Description
`data`	`numpy.ndarray` or `dask.array` holding the array's values.
`dims`	dimension names for each axis. For example:(`x` , `y` , `z`) (`lat` , `lon` , `time`).
`coords`	a dict-like container of arrays (coordinates) that label each point (e.g., 1-dimensional arrays of numbers, datetime objects or strings)
`attrs`	an `OrderedDict` to hold arbitrary attributes/metadata (such as units)
`name`	an arbitrary name of the array

Subsetting a dataset by time and space (Slicing and Dicing)

The index selection is equivalent to using `isel` like so

In [140…

```
stl1.isel(time=0)
```

Out[140]:

xarray.DataArray 'stl1' (**latitude**: 37, **longitude**: 33)

```
array([[302.52182, 301.0825 , 301.4906 , ..., 303.57666, 303.59207, 303.166
4 ],
       [301.12918, 300.25824, 301.25815, ..., 303.9028 , 303.92615, 303.404
72],
       [301.2425 , 300.92432, 301.44363, ..., 303.60983, 303.9848 , 303.451
42],
       ...,
       [301.5413 , 301.5628 , 301.57846, ..., 301.5023 , 301.46725, 301.469
12],
       [301.61374, 301.61374, 301.60577, ..., 301.58243, 301.5336 , 301.506
26],
       [301.66257, 301.6233 , 301.59595, ..., 301.69177, 301.63126, 301.576
6 ]],
      dtype=float32)
```

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	()	datetime64[ns]	1980-01-01

▾ Attributes:

units : K

long_name : Soil temperature level 1

standard_name : surface_temperature

In [ ]:

In [ ]:

The power of xarray comes with the close association of data with coordinates. So it is possible to use the equivalent `.sel` operator but with coordinate values. For example, to select an area that includes the Indian Ocean and Australia use `slice` to indicate the range of latitude and longitude values required and pass as key/value pairs to `sel`. `slice` will include coordinate values less than **or equal** to the upper bound, not like `range` in basic python that excludes the upper bound

In [141…

```
stl1.sel(longitude=slice(-5,-1),latitude=slice(12,10))
```

Out[141]: xarray.DataArray 'stl1' (**time**: 504, **latitude**: 9, **longitude**: 17)

```
array([[[302.52182, 301.0825 , ..., 301.7504 , 302.12537],
        [301.12918, 300.25824, ..., 302.06088, 302.20737],
        ...,
        [301.95926, 302.24637, ..., 303.729  , 303.807  ],
        [301.67212, 302.10388, ..., 303.09796, 304.27567]],

       [[303.71335, 302.95758, ..., 302.75244, 303.2739 ],
        [303.41055, 302.80313, ..., 302.97906, 303.20755],
        ...,
        [304.85788, 305.21347, ..., 306.36172, 306.37552],
        [304.40488, 304.68988, ..., 305.26016, 306.63928]],

       ...,

       [[301.82578, 301.57977, ..., 303.5719 , 303.86884],
        [301.77905, 301.36484, ..., 303.36093, 303.72235],
        ...,
        [301.9821 , 302.18137, ..., 304.7456 , 304.45477],
        [301.9566 , 302.1267 , ..., 303.73987, 304.84726]],

       [[300.71628, 300.56393, ..., 300.73193, 301.12625],
        [300.28452, 300.14972, ..., 300.9466 , 301.10477],
        ...,
        [300.42914, 300.97607, ..., 302.1986 , 302.14792],
        [300.552  , 300.8256 , ..., 301.2924 , 302.5014 ]]], dtype=float32)
```

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... -1.5 -1.25 -1.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 10.5 10.25 10.0
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▾ Attributes:

units :	K
long_name :	Soil temperature level 1
standard_name :	surface_temperature

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [142…

```
stl1.sel(time='1980-01-01', longitude=slice(-5,-1),latitude=slice(12,10))
```

Out[142]: xarray.DataArray 'stl1' (**latitude**: 9, **longitude**: 17)

```
array([[302.52182, 301.0825 , 301.4906 , 302.41437, 301.91043, 302.07257,
        302.84427, 303.28555, 302.86948, 301.8226 , 301.9282 , 302.0081 ,
        302.12537, 302.0102 , 301.45346, 301.7504 , 302.12537],
       [301.12918, 300.25824, 301.25815, 301.9144 , 301.2661 , 302.05902,
        302.92227, 303.3734 , 302.9183 , 302.59613, 302.8774 , 303.0844 ,
        302.72696, 302.438  , 302.1352 , 302.06088, 302.20737],
       [301.2425 , 300.92432, 301.44363, 301.72095, 302.30716, 302.7113 ,
        302.78934, 303.2171 , 303.29324, 303.23276, 303.47107, 303.22876,
        302.516  , 302.2562 , 302.00223, 302.3382 , 302.48865],
       [300.7152 , 300.39093, 300.9397 , 302.13306, 303.08228, 303.4124 ,
        303.479  , 303.08044, 303.1468 , 303.42407, 303.45938, 302.7503 ,
        302.05106, 302.1527 , 302.44965, 302.77765, 302.74844],
       [299.64285, 299.47488, 300.56473, 302.50616, 302.95358, 303.35986,
        303.34024, 302.9398 , 302.73465, 303.0101 , 303.11945, 302.78348,
        302.25223, 302.85382, 303.20145, 303.4769 , 303.4477 ],
       [300.72504, 300.78738, 301.62543, 302.2992 , 302.3305 , 302.8188 ,
        302.78748, 302.70944, 302.7113 , 302.40082, 302.84796, 303.43973,
        302.81665, 303.30307, 303.66452, 303.9575 , 303.8147 ],
       [302.76996, 302.51202, 302.4085 , 301.93988, 301.85205, 302.48282,
        302.7113 , 302.5335 , 302.50616, 302.4653 , 302.73465, 303.14493,
        302.60782, 303.37524, 303.73483, 303.80304, 303.8187 ],
       [301.95926, 302.24637, 302.6293 , 301.5726 , 301.47098, 301.93774,
        302.555  , 303.2816 , 303.58835, 302.8323 , 301.99454, 303.5042 ,
        303.928  , 303.51404, 303.53738, 303.729  , 303.807  ],
       [301.67212, 302.10388, 302.59216, 301.00418, 300.3421 , 302.0922 ,
        303.16058, 303.41824, 303.56473, 303.27972, 301.77588, 303.39676,
        303.73856, 302.23682, 301.4516 , 303.09796, 304.27567]], dtype=floa
t32)
```

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... -1.5 -1.25 -1.0	
latitude	(latitude)	float32	12.0 11.75 11.5 ... 10.5 10.25 10.0	
time	()	datetime64[ns]	1980-01-01	

▾ Attributes:

units :	K
long_name :	Soil temperature level 1
standard_name :	surface_temperature

In [ ]:

In [ ]:

In [ ]:

In [ ]:

It is also possible to use `slice` for the `time` dimension. To select Mar to November of 1980:

In [143…
```
stl1.sel(time=slice('1980-03','1980-11'), longitude=slice(-5,-1),latitude=slice(12,10))
```

Out[143]: xarray.DataArray 'stl1' (**time**: 9, **latitude**: 9, **longitude**: 17)

```
array([[[307.20108, 306.332  , ..., 305.8575 , 306.46283],
        [306.83194, 306.32217, ..., 306.33597, 306.39062],
        ...,
        [307.252  , 307.63467, ..., 309.89642, 309.81442],
        [306.25583, 306.66025, ..., 308.24026, 309.89853]],

       [[308.43796, 307.8207 , ..., 308.65875, 309.0767 ],
        [307.87936, 307.81116, ..., 308.81503, 308.71738],
        ...,
        [306.9869 , 306.95374, ..., 309.3657 , 309.5395 ],
        [305.92438, 306.03955, ..., 307.33243, 309.24283]],

       ...,

       [[302.06964, 301.59702, ..., 303.92297, 304.41736],
        [301.62622, 301.34094, ..., 302.94455, 303.21817],
        ...,
        [300.40738, 300.63797, ..., 301.64957, 301.49353],
        [300.28238, 300.19455, ..., 300.78632, 301.4269 ]],

       [[303.45938, 302.22913, ..., 303.0024 , 303.25824],
        [302.7564 , 302.7291 , ..., 302.477  , 302.38916],
        ...,
        [300.76218, 301.66656, ..., 303.0414 , 302.44778],
        [300.65283, 300.25452, ..., 301.73474, 302.60013]]], dtype=float32)
```

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... -1.5 -1.25 -1.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 10.5 10.25 10.0
time	(time)	datetime64[ns]	1980-03-01 ... 1980-11-01

▾ Attributes:

units :	K
long_name :	Soil temperature level 1
standard_name :	surface_temperature

The `slice` operator selects values between an upper and lower bound. If a single coordinate value is required when using `sel` it must either correspond to an *exact* value in the coordinate array, or the `method` argument specified to tell xarray how to choose a value.

In [ ]:

In [ ]:

In [144…
```
stl1.sel(latitude=11.4, longitude=2.1, method='nearest')
```

Out[144]: xarray.DataArray 'stl1' (**time**: 504)

```
array([304.53943, 305.9496 , 308.8437 , ..., 304.54926, 304.53094, 302.5229
],
      dtype=float32)
```

▾ Coordinates:

longitude	()	float32	2.0
latitude	()	float32	11.5
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▾ Attributes:

units :	K
long_name :	Soil temperature level 1
standard_name :	surface_temperature

So the closest location in the data was at `lat=11.5`, `lon=2.0`.

Calculating metrics

xarray is built on top of numpy, which means it implements many of the numpy operators as native methods, and those it doesn't can still be used on the underlying numpy arrays contained with an `xarray.DataArray` object.

It is straightforward to calculate the mean temperature for all locations and times in the data

In [145…
```
stl1.mean()
```

Out[145]: xarray.DataArray 'stl1'

```
array(301.1962, dtype=float32)
```

▸ Coordinates: (0)

▸ Attributes: (0)

It is possible to specify a dimension along which to compute an operator. For example, to calculate the mean in time for all locations specify the `time` dimension as the dimension along which the mean should be calculated:

In [146…
```
stl1.mean(dim='time')
```

Out[146]: xarray.DataArray 'stl1' (**latitude**: 37, **longitude**: 33)

```
array([[303.37967, 302.97678, 302.95703, ..., 304.65826, 304.96573,
        304.8317 ],
       [303.01404, 302.86688, 303.10373, ..., 304.66656, 304.87384,
        304.66583],
       [302.67996, 302.79303, 302.90735, ..., 304.22144, 304.64966,
        304.42004],
       ...,
       [301.0193 , 301.02335, 301.0225 , ..., 301.12622, 301.1267 ,
```

```
              301.12497],
             [301.0628 , 301.0594 , 301.0604 , ..., 301.14273, 301.13943,
              301.1399 ],
             [301.0849 , 301.0725 , 301.067  , ..., 301.1616 , 301.14304,
              301.144  ]], dtype=float32)
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0

► Attributes: (0)

It is common to calculate a 30-year climatology, which is simple using `sel` and chaining operators

In [147…

```
stl1_clim = stl1.sel(time=slice('1960-01','1989-12')).mean(dim='time')
stl1_clim
```

Out[147]: xarray.DataArray 'stl1' (**latitude**: 37, **longitude**: 33)

```
array([[303.24066, 302.66928, 302.6544 , ..., 304.25732, 304.60577,
        304.49542],
       [302.76706, 302.5581 , 302.79788, ..., 304.3579 , 304.58496,
        304.38937],
       [302.4606 , 302.46518, 302.54684, ..., 303.94223, 304.44708,
        304.21347],
       ...,
       [300.83337, 300.85507, 300.86224, ..., 300.97415, 300.9727 ,
        300.97205],
       [300.8954 , 300.8989 , 300.89853, ..., 300.99042, 300.98047,
        300.979  ],
       [300.9539 , 300.92972, 300.91168, ..., 301.01544, 300.98605,
        300.98376]], dtype=float32)
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0

► Attributes: (0)

In [148…

```
sst_Celsius = stl1 - 273.15
sst_Celsius
```

Out[148]: xarray.DataArray 'stl1' (**time**: 504, **latitude**: 37, **longitude**: 33)

```
array([[[29.371826, 27.932495, 28.340607, ..., 30.426666, 30.442078,
         30.016418],
        [27.979187, 27.108246, 28.108154, ..., 30.752808, 30.776154,
         30.25473 ],
        [28.092499, 27.774323, 28.29364 , ..., 30.459839, 30.834808,
         30.301422],
        ...,
        [28.391296, 28.412811, 28.428467, ..., 28.352295, 28.31726 ,
         28.319122],
        [28.463745, 28.463745, 28.45578 , ..., 28.432434, 28.383606,
         28.356262],
        [28.512573, 28.473297, 28.445953, ..., 28.541779, 28.481262,
         28.426605]],
```

```
        [[30.563354, 29.807587, 30.16504 , ..., 31.748993, 31.868164,
          31.534058],
         [30.26056 , 29.653137, 30.473633, ..., 32.155273, 32.233307,
          31.981476],
         [30.487183, 30.368042, 30.862152, ..., 32.05179 , 32.405273,
          32.12982 ],
 ...
         [28.503815, 28.51178 , 28.515747, ..., 28.742126, 28.722748,
          28.71106 ],
         [28.458984, 28.47464 , 28.466919, ..., 28.689575, 28.687439,
          28.683746],
         [28.390778, 28.41809 , 28.415985, ..., 28.619263, 28.625092,
          28.621124]],

        [[27.566284, 27.41394 , 27.823944, ..., 28.400055, 28.390503,
          28.152191],
         [27.134521, 26.999725, 27.718597, ..., 28.878784, 28.65799 ,
          28.212708],
         [27.005554, 27.136627, 27.718597, ..., 28.60147 , 28.751678,
          28.310364],
         ...,
         [29.003784, 29.032959, 29.021301, ..., 29.011475, 28.985992,
          28.956818],
         [28.913818, 28.958649, 28.98996 , ..., 28.978302, 28.97644 ,
          28.958649],
         [28.85144 , 28.878784, 28.906128, ..., 29.005615, 28.960785,
          28.935303]]], dtype=float32)
```

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0	
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0	
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01	

▸ Attributes: (0)

Grouping and resampling

xarray was developed as an n-dimensional extension of pandas, a very powerful data analysis library designed primarily for tabular data and time series analysis.

As a result xarray can utilise much of the time series manipulation power of pandas. The relevant xarray documentation is contained in the `GroupBy` and `Time series data` sections.

Xarray has some very useful high level objects that let you do common computations:

- `groupby` : Bin data in to groups and reduce
- `resample` : Groupby specialized for time axes. Either downsample or upsample your data.

groupby

In [149…

```
# seasonal groups
ds.groupby("time.season")
```

Out[149]:

```
DatasetGroupBy, grouped over 'season'
4 groups with labels 'DJF', 'JJA', 'MAM', 'SON'.
```

In [150…
```python
# day of the week groups
ds.groupby("time.dayofweek")
```

Out[150]:
```
DatasetGroupBy, grouped over 'dayofweek'
7 groups with labels 0, 1, 2, 3, 4, 5, 6.
```

In [151…
```python
# monthly groups
ds.groupby("time.month")
```

Out[151]:
```
DatasetGroupBy, grouped over 'month'
12 groups with labels 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.
```

In [152…
```python
# yearly groups
ds.groupby("time.year")
```

Out[152]:
```
DatasetGroupBy, grouped over 'year'
42 groups with labels 1980, 1981, 1982, ..., 2020, 2021.
```

In [153…
```python
# compute a seasonal mean
seasonal_mean = ds.groupby("time.season").mean()
seasonal_mean
```

Out[153]: xarray.Dataset

▸ Dimensions: (**longitude**: 33, **latitude**: 37, **season**: 4)

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
season	(season)	object	'DJF' 'JJA' 'MAM' 'SON'

▾ Data variables:

stl1	(season, latitude, longitude)	float32	301.6 301.3 301.4 ... 300.3 300.3
stl2	(season, latitude, longitude)	float32	301.1 300.7 300.8 ... 300.3 300.3
stl3	(season, latitude, longitude)	float32	301.1 300.7 300.7 ... 300.3 300.3
stl4	(season, latitude, longitude)	float32	301.4 300.9 300.9 ... 300.3 300.3
swvl1	(season, latitude, longitude)	float32	0.1169 0.1173 ... 4.969e-07
swvl2	(season, latitude, longitude)	float32	0.1451 0.1505 ... -3.926e-08
swvl3	(season, latitude, longitude)	float32	0.1652 0.1777 ... 6.866e-07
swvl4	(season, latitude, longitude)	float32	0.1763 0.1949 0.1905 ... 0.0 0.0

▸ Attributes: (0)

resample

In [154…
```python
# resample to monthly frequency
stl1.resample(time="M").mean()
```

Out[154]: xarray.DataArray 'stl1' (**time**: 504, **latitude**: 37, **longitude**: 33)

```
array([[[302.52182, 301.0825 , 301.4906 , ..., 303.57666, 303.59207,
         303.1664 ],
        [301.12918, 300.25824, 301.25815, ..., 303.9028 , 303.92615,
         303.40472],
        [301.2425 , 300.92432, 301.44363, ..., 303.60983, 303.9848 ,
         303.45142],
        ...,
```

```
       [301.5413 , 301.5628 , 301.57846, ..., 301.5023 , 301.46725,
        301.46912],
       [301.61374, 301.61374, 301.60577, ..., 301.58243, 301.5336 ,
        301.50626],
       [301.66257, 301.6233 , 301.59595, ..., 301.69177, 301.63126,
        301.5766 ]],

      [[303.71335, 302.95758, 303.31503, ..., 304.899  , 305.01816,
        304.68405],
       [303.41055, 302.80313, 303.62363, ..., 305.30527, 305.3833 ,
        305.13147],
       [303.63718, 303.51804, 304.01215, ..., 305.20178, 305.55527,
        305.27982],
...
       [301.6538 , 301.66177, 301.66574, ..., 301.89212, 301.87274,
        301.86105],
       [301.60898, 301.62463, 301.6169 , ..., 301.83957, 301.83743,
        301.83374],
       [301.54077, 301.56808, 301.56598, ..., 301.76926, 301.7751 ,
        301.77112]],

      [[300.71628, 300.56393, 300.97394, ..., 301.55005, 301.5405 ,
        301.3022 ],
       [300.28452, 300.14972, 300.8686 , ..., 302.02878, 301.80798,
        301.3627 ],
       [300.15555, 300.28662, 300.8686 , ..., 301.75146, 301.90167,
        301.46036],
       ...,
       [302.15378, 302.18295, 302.1713 , ..., 302.16147, 302.136  ,
        302.1068 ],
       [302.0638 , 302.10864, 302.13995, ..., 302.1283 , 302.12643,
        302.10864],
       [302.00143, 302.02878, 302.05612, ..., 302.1556 , 302.11078,
        302.0853 ]]], dtype=float32)
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0	
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0	
time	(time)	datetime64[ns]	1980-01-31 ... 2021-12-31	

▶ Attributes: (0)

In [155…

```
# resample to yearly frequency
a=stl1.resample(time="Y").mean()
a
```

Out[155]: xarray.DataArray 'stl1' (**time**: 42, **latitude**: 37, **longitude**: 33)

```
array([[[303.18253, 302.4697 , 302.5018 , ..., 304.3315 , 304.59042,
         304.33963],
        [302.63126, 302.34338, 302.51022, ..., 304.47977, 304.554  ,
         304.33542],
        [302.50366, 302.59158, 302.50433, ..., 303.99014, 304.39386,
         304.09418],
        ...,
        [300.7548 , 300.76767, 300.7667 , ..., 300.87057, 300.86072,
         300.85724],
```

```
        [300.8279 , 300.8172 , 300.80057, ..., 300.89627, 300.8756 ,
         300.86243],
        [300.87253, 300.8325 , 300.79663, ..., 300.928  , 300.8906 ,
         300.87076]],

       [[303.1794 , 302.7087 , 302.64505, ..., 303.8895 , 304.19028,
         303.97037],
        [302.62177, 302.54477, 302.7523 , ..., 303.86624, 304.11478,
         303.95343],
        [302.31387, 302.462  , 302.4935 , ..., 303.45822, 303.94437,
         303.7453 ],
...
        [301.4413 , 301.4462 , 301.4465 , ..., 301.35684, 301.3422 ,
         301.3402 ],
        [301.42667, 301.43756, 301.4496 , ..., 301.3757 , 301.36124,
         301.35422],
        [301.38907, 301.40585, 301.4147 , ..., 301.39447, 301.3692 ,
         301.36334]],

       [[303.5862 , 303.32864, 303.2825 , ..., 305.2783 , 305.6195 ,
         305.34567],
        [303.28244, 303.03674, 303.4528 , ..., 305.07175, 305.38574,
         305.306  ],
        [302.97427, 303.06442, 303.40543, ..., 304.6406 , 305.09442,
         304.9858 ],
        ...,
        [301.59604, 301.6074 , 301.605  , ..., 301.67642, 301.65738,
         301.64243],
        [301.6056 , 301.61407, 301.6271 , ..., 301.67004, 301.65408,
         301.64813],
        [301.5934 , 301.60516, 301.60916, ..., 301.6732 , 301.64322,
         301.63068]]], dtype=float32)
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	(time)	datetime64[ns]	1980-12-31 ... 2021-12-31

▸ Attributes: (0)

16. Introduction to Cartopy

Overview

1. Basic concepts: map projections and GeoAxes georeferenced GeoAxes
2. Explore some of Cartopy's map projections Create a map with a specified projection
3. Create regional maps

This tutorial will lead you through some basics of creating maps with specified projections with Cartopy, and adding geographic features like coastlines and borders.

Later tutorials will focus on how to plot data on map projections.

In [156…

```
import cartopy.crs as ccrs
```

```
import cartopy.feature as cfeature
import matplotlib.pyplot as plt
import xarray as xr
```

Basic concepts: map projections and GeoAxes

- Extend Matplotlib's axes into georeferenced GeoAxes Recall that in Matplotlib, what we might tradtionally term a figure consists of two key components: a figure and an associated subplot axes instance.

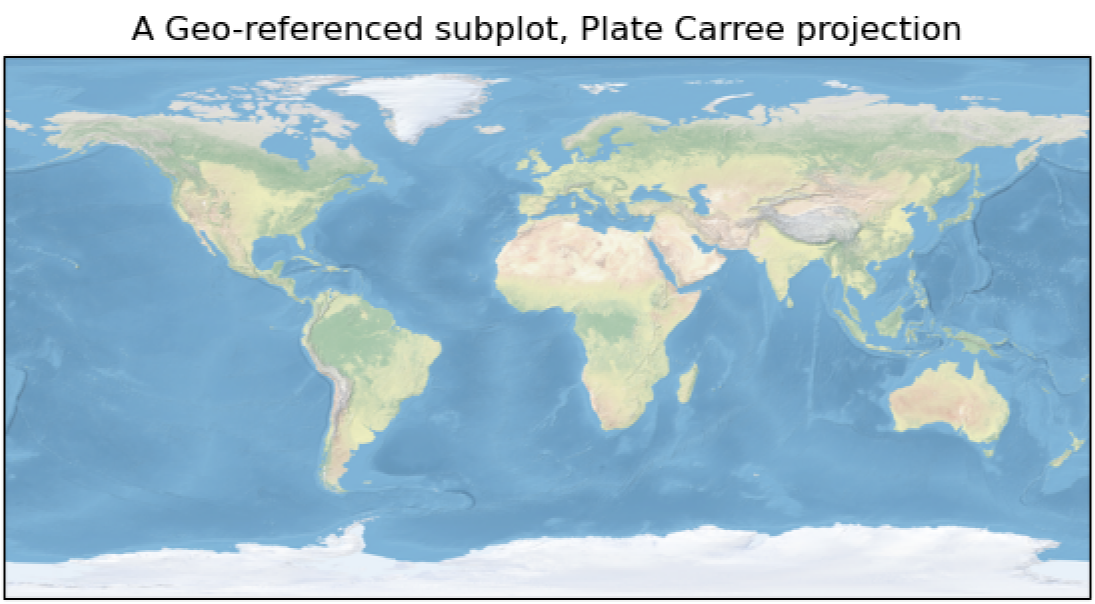
By virtue of importing Cartopy, we can now convert the axes into a GeoAxes by specifying a projection that we have imported from Cartopy's Coordinate Reference System class as ccrs . This will effectively georeference the subplot.

Create a map with a specified projection

Here we'll create a GeoAxes that uses the PlateCarree projection.

In [157…
```
# Initialize the figure
fig = plt.figure(figsize=(7,5))

# use the PlateCarree projection
ax = plt.subplot(1, 1, 1, projection=ccrs.PlateCarree())

#set title
ax.set_title("A Geo-referenced subplot, Plate Carree projection");

# Add feature to the map
ax.stock_img()
```

Out[157]:
```
<matplotlib.image.AxesImage at 0x7fa0cbe94520>
```



The next few lines of code go through the basic process of creating a map:

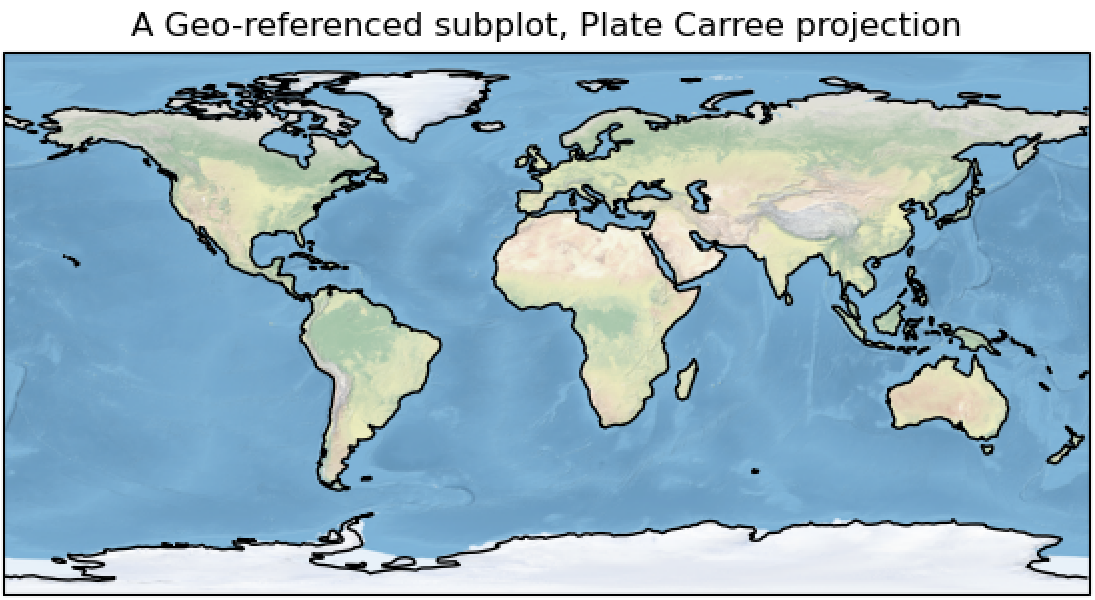
1. Initialize the map and specify the size of the figure with the figsize argument
2. Add a subplot which specifies the projection used
3. Add features to the subplot

With only a few lines of codes, the bare minimum for a map is created. Unfortunately, this map is rather bland, so we'll add some extra features to it.

Although the figure seems empty, it has in fact been georeferenced, using one of Cartopy's map projections that is provided by Cartopy's crs (coordinate reference system) class. We can now add in cartographic features, in the form of shapefiles, to our subplot. One of them is coastlines , which is a callable GeoAxes method that can be plotted directly on our subplot.

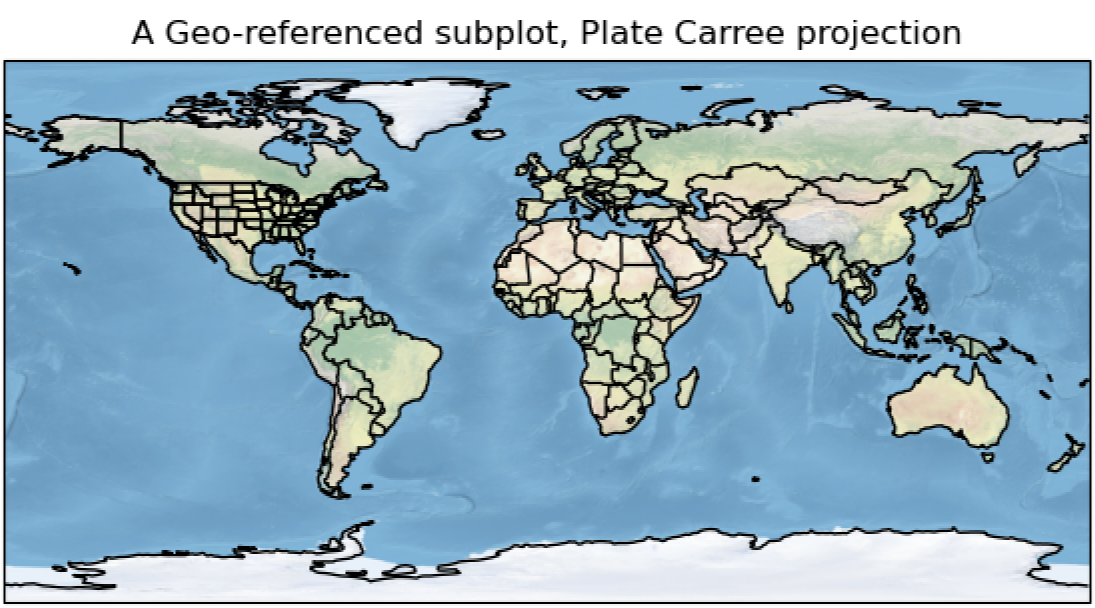
In [158…
```python
fig = plt.figure(figsize=(7, 5))
ax = plt.subplot(1, 1, 1, projection=ccrs.PlateCarree())
ax.set_title("A Geo-referenced subplot, Plate Carree projection");
ax.add_feature(cfeature.COASTLINE)
ax.stock_img()
```

Out[158]:
```
<matplotlib.image.AxesImage at 0x7fa0f80c0490>
```



In [159…
```python
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(1, 1, 1, projection=ccrs.PlateCarree())
ax.set_title("A Geo-referenced subplot, Plate Carree projection");
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.STATES)
ax.stock_img()
```

Out[159]:
```
<matplotlib.image.AxesImage at 0x7fa0cbe02a60>
```



By modifying the variable ax, more information is added to the map. In this case, national borders and coastlines are drawn. Additional features, such as major rivers and lakes map also be included with the same method. A full list of features can be found here.

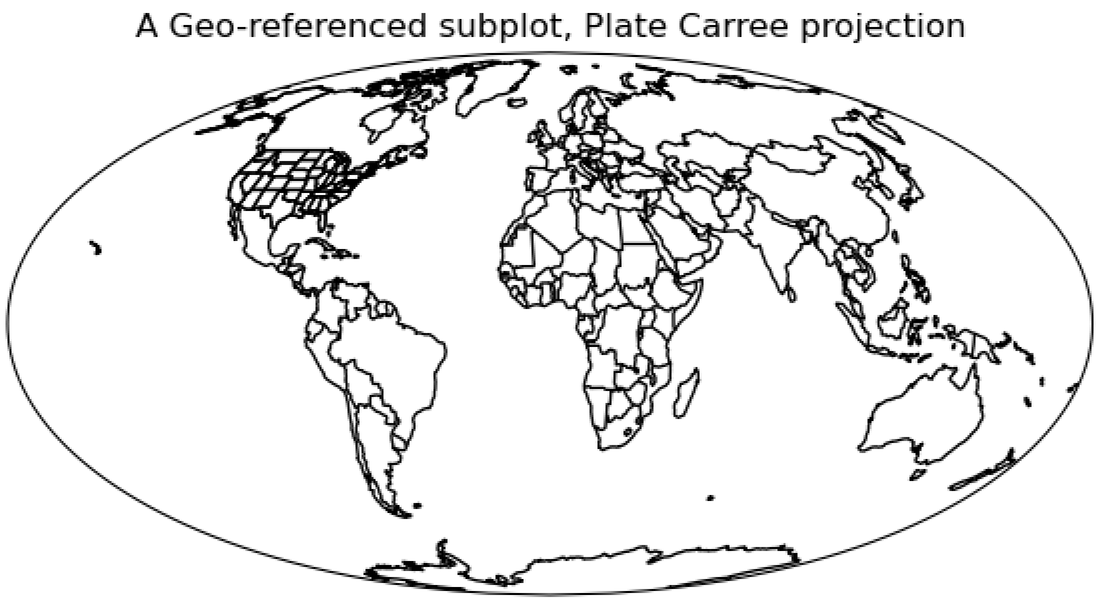
Different Projections

In [160…

```python
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(1, 1, 1, projection=ccrs.Mollweide())
ax.set_title("A Geo-referenced subplot, Plate Carree projection");
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.STATES)
#ax.stock_img()
```

Out[160]:

```
<cartopy.mpl.feature_artist.FeatureArtist at 0x7fa0cbd44e20>
```

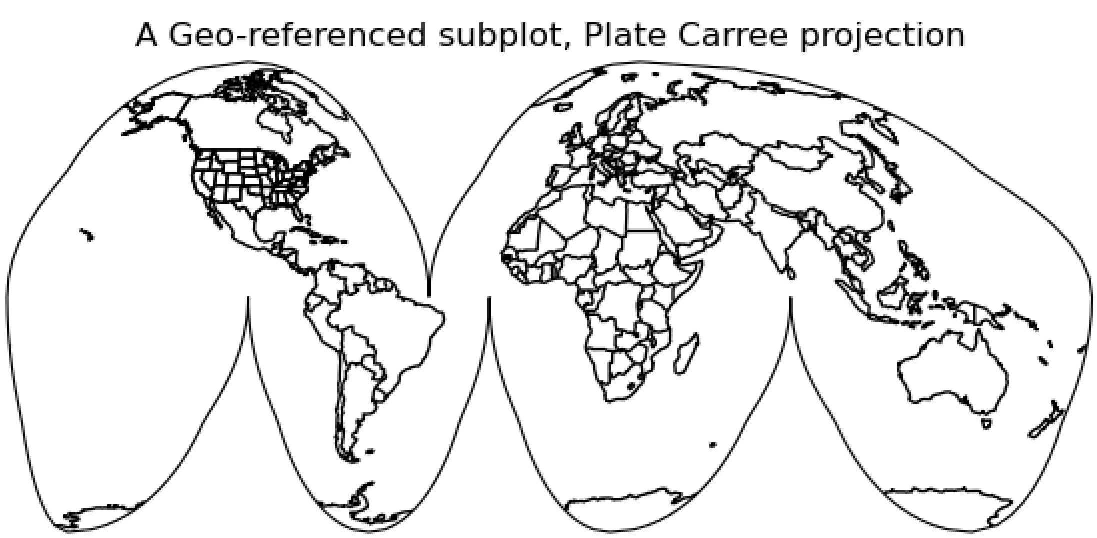


In [161…

```python
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(1, 1, 1, projection=ccrs.InterruptedGoodeHomolosine())
ax.set_title("A Geo-referenced subplot, Plate Carree projection");
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.STATES)
#ax.stock_img()
```

Out[161]:

```
<cartopy.mpl.feature_artist.FeatureArtist at 0x7fa0cbcfedc0>
```



In [162…

```python
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(1, 1, 1, projection=ccrs.LambertAzimuthalEqualArea())
ax.set_title("A Geo-referenced subplot, Plate Carree projection");
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.STATES)
```

Out[162]:

```
<cartopy.mpl.feature_artist.FeatureArtist at 0x7fa0cbc4cd90>
```

A Geo-referenced subplot, Plate Carree projection

Create regional maps

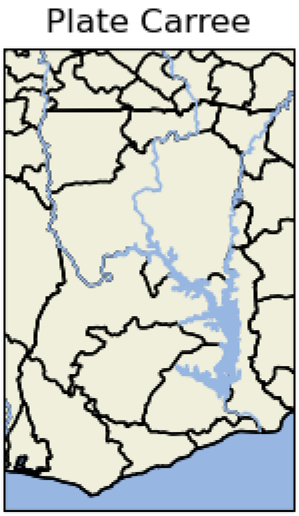
Now, let's go back to PlateCarree, but let's use Cartopy's set_extent method to restrict the map coverage to a North American view. Let's also choose a lower resolution for coastlines, just to illustrate how one can specify that. Plot lat/lon lines as well

In [163…

```python
fig = plt.figure(figsize=(4, 3))
ax = plt.subplot(1, 1, 1, projection=ccrs.PlateCarree())
ax.set_title('Plate Carree')
ax.set_extent([-3.5, 1.2, 4.5, 12], crs=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.LAND)
ax.add_feature(cfeature.OCEAN)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
```

Out[163]:

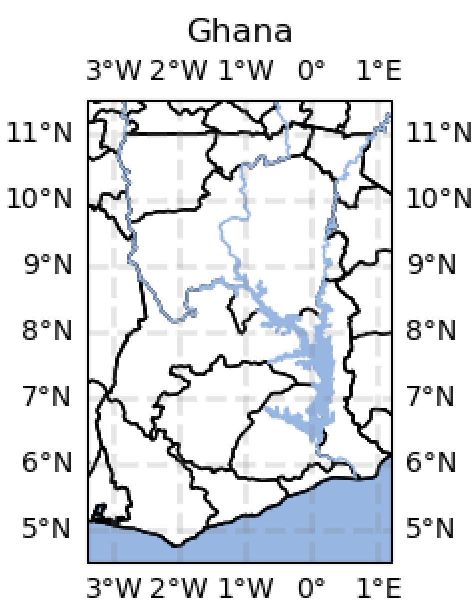
```
<cartopy.mpl.feature_artist.FeatureArtist at 0x7fa0cbbd5460>
```




```
Note that in the set_extent call, we specified PlateCarree. This ensures that
the values we passed
into set_extent will be transformed from degrees into the values appropriate
for the projection we use
for the map.
```

In [164…

```
fig = plt.figure(figsize=(4, 3))
ax = plt.subplot(1, 1, 1, projection=ccrs.PlateCarree())
ax.set_title('Ghana')
ax.set_extent([-3.4, 1.2, 4.5, 11.5], crs=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.OCEAN)
gl = ax.gridlines(draw_labels=True, linewidth=2, color='gray', alpha=0.2, linestyle='--'
```



Summary

1. Cartopy allows for georeferencing Matplotlib axes objects.
2. Cartopy's crs class supports a variety of map projections.
3. Cartopy's feature class allows for a variety of cartographic features to be overlaid on the figure.

Cartopy: Plotting on a map (Spatial plots)

In [166…
```
#open our data
Data=xr.open_dataset('SOIL_MOISTURE.nc')
Data
```

Out[166]: xarray.Dataset

► Dimensions: (**longitude**: 33, **latitude**: 37, **time**: 504)

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▼ Data variables:

stl1	(time, latitude, longitude)	float32	...
stl2	(time, latitude, longitude)	float32	...
stl3	(time, latitude, longitude)	float32	...
stl4	(time, latitude, longitude)	float32	...
swvl1	(time, latitude, longitude)	float32	...
swvl2	(time, latitude, longitude)	float32	...
swvl3	(time, latitude, longitude)	float32	...
swvl4	(time, latitude, longitude)	float32	...

▼ Attributes:

Conventions : CF-1.6

history : 2022-07-08 10:10:30 GMT by grib_to_netcdf-2.25.1: /opt/ecmwf/mars-client/bin/grib_to_netcdf.bin -S param -o /cache/data9/adaptor.mars.internal-1657275023.2669723-2621-7-23d9a0bf-7bf2-4764-8a86-5d53fd492b90.nc /cache/tmp/23d9a0bf-7bf2-4764-8a86-5d53fd492b90-adaptor.mars.internal-1657274477.0104525-2621-5-tmp.grib

In [167…
```
#subsetting (choosing a variable to work with)
swvl1=Data['swvl1']
swvl1
```

Out[167]: xarray.DataArray 'swvl1' (**time**: 504, **latitude**: 37, **longitude**: 33)

```
[615384 values with dtype=float32]
```

▼ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
time	(time)	datetime64[ns]	1980-01-01 ... 2021-12-01

▼ Attributes:

units : m**3 m**-3

long_name : Volumetric soil water layer 1

In [168…
```
#monthly climatology
swvl1_clim=swvl1.groupby('time.month').mean('time')
swvl1_clim
```

Out[168]: xarray.DataArray 'swvl1' (**month**: 12, **latitude**: 37, **longitude**: 33)

```
array([[[ 1.18346043e-01,  1.18534110e-01,  1.17789485e-01, ...,
          1.02457412e-01,  1.00163572e-01,  1.43525571e-01],
        [ 1.18844658e-01,  1.18529059e-01,  1.17371470e-01, ...,
          9.80118662e-02,  9.70092863e-02,  9.70441401e-02],
        [ 1.19557507e-01,  1.19243510e-01,  1.18344106e-01, ...,
          9.66895297e-02,  9.65574905e-02,  9.59405750e-02],
        ...,
        [ 1.70121587e-06,  1.70121587e-06,  1.70121587e-06, ...,
          1.70121587e-06,  1.70121587e-06,  1.70121587e-06],
        [ 1.70121587e-06,  1.70121587e-06,  1.70121587e-06, ...,
          1.70121587e-06,  1.70121587e-06,  1.70121587e-06],
        [ 1.70121587e-06,  1.70121587e-06,  1.70121587e-06, ...,
          1.70121587e-06,  1.70121587e-06,  1.70121587e-06]],

       [[ 1.12452619e-01,  1.12692729e-01,  1.11940496e-01, ...,
          9.40309390e-02,  9.13064107e-02,  1.33338884e-01],
        [ 1.11701883e-01,  1.11704938e-01,  1.10791571e-01, ...,
          8.76996368e-02,  8.70967954e-02,  8.62170607e-02],
        [ 1.12270355e-01,  1.11703850e-01,  1.11204170e-01, ...,
          8.60709101e-02,  8.59178901e-02,  8.43878090e-02],
...
          1.15980708e-06,  1.15980708e-06,  1.15980708e-06],
        [ 1.15980708e-06,  1.15980708e-06,  1.15980708e-06, ...,
          1.15980708e-06,  1.15980708e-06,  1.15980708e-06],
        [ 1.15980708e-06,  1.15980708e-06,  1.15980708e-06, ...,
          1.15980708e-06,  1.15980708e-06,  1.15980708e-06]],

       [[ 1.19942963e-01,  1.20822191e-01,  1.19079292e-01, ...,
          1.07859671e-01,  1.05180852e-01,  1.50091007e-01],
        [ 1.21216886e-01,  1.21381283e-01,  1.19487561e-01, ...,
          1.04116857e-01,  1.02500044e-01,  1.02601595e-01],
        [ 1.21966057e-01,  1.22442618e-01,  1.21496901e-01, ...,
          1.02393091e-01,  1.02086559e-01,  1.01747639e-01],
        ...,
        [ 1.88251340e-06,  1.88251340e-06,  1.88251340e-06, ...,
          1.88251340e-06,  1.88251340e-06,  1.88251340e-06],
        [ 1.88251340e-06,  1.88251340e-06,  1.88251340e-06, ...,
          1.88251340e-06,  1.88251340e-06,  1.88251340e-06],
        [ 1.88251340e-06,  1.88251340e-06,  1.88251340e-06, ...,
          1.88251340e-06,  1.88251340e-06,  1.88251340e-06]]],
      dtype=float32)
```

▾ Coordinates:

longitude	(longitude)	float32	-5.0 -4.75 -4.5 ... 2.5 2.75 3.0
latitude	(latitude)	float32	12.0 11.75 11.5 ... 3.5 3.25 3.0
month	(month)	int64	1 2 3 4 5 6 7 8 9 10 11 12

▸ Attributes: (0)

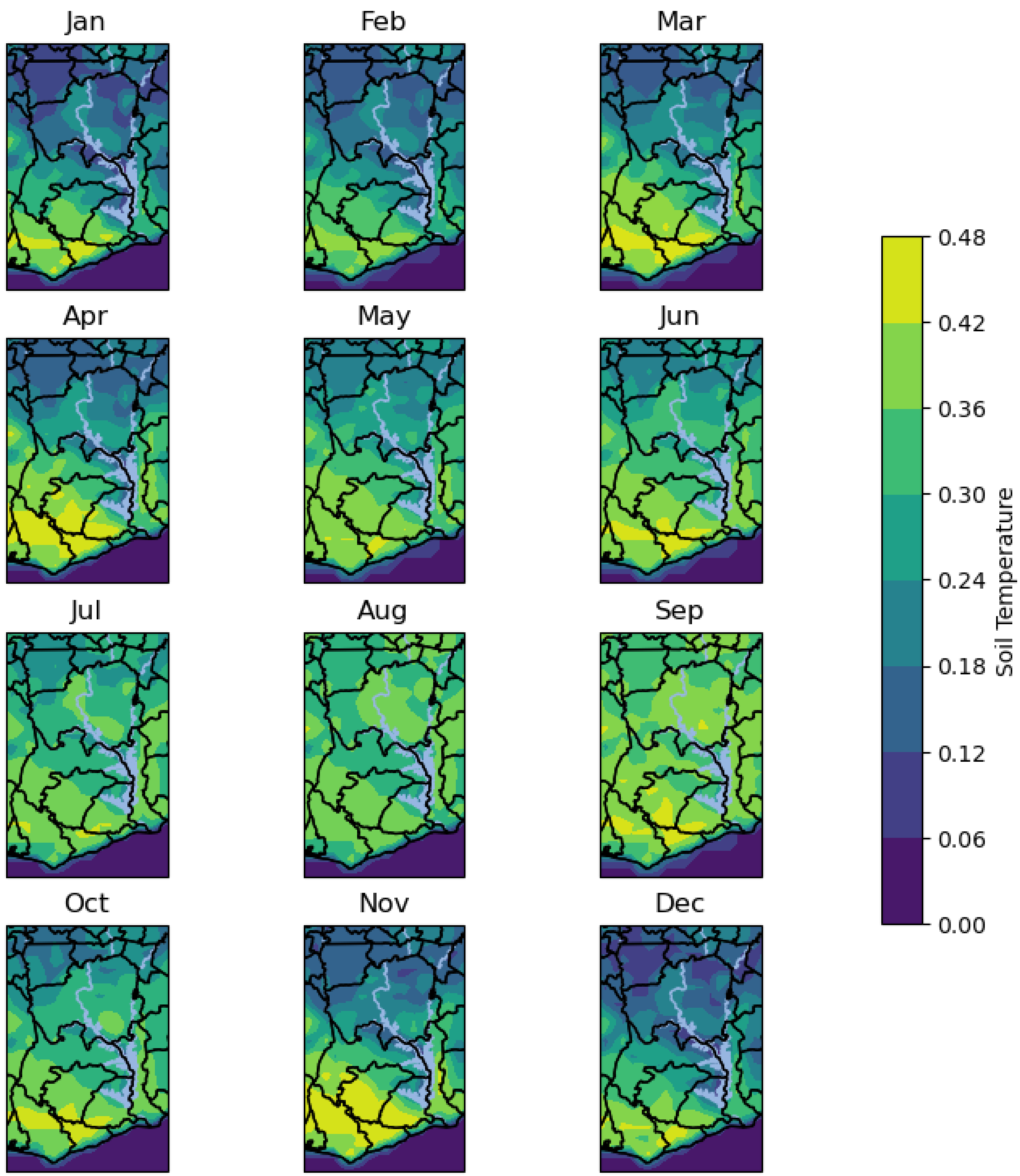
In [172…

```
fig=plt.figure(figsize=(10,8.5))
month_names=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
ax=plt.subplot(4,3,1,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
```

```
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[0],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[0])

ax=plt.subplot(4,3,2,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[1],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[1])

ax=plt.subplot(4,3,3,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[2],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[2])

ax=plt.subplot(4,3,4,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[3],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[3])

#--------------------------------------------------------

ax=plt.subplot(4,3,5,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[4],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[4])

ax=plt.subplot(4,3,6,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[5],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[5])

ax=plt.subplot(4,3,7,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
```

```
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[6],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[6])

ax=plt.subplot(4,3,8,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[7],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[7])

#---------------------------------------------------------

ax=plt.subplot(4,3,9,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[8],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[8])

ax=plt.subplot(4,3,10,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[9],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[9])

ax=plt.subplot(4,3,11,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[10],
            transform=ccrs.PlateCarree())
ax.set_title(month_names[10])

ax=plt.subplot(4,3,12,  projection=ccrs.PlateCarree())
ax.add_feature(cfeature.COASTLINE)
ax.add_feature(cfeature.RIVERS)
ax.add_feature(cfeature.LAKES)
ax.add_feature(cfeature.STATES)
ax.add_feature(cfeature.OCEAN)
ax.set_extent([-3.4,1.2,4.5,11.5], crs=ccrs.PlateCarree())
cb=ax.contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[11],
               transform=ccrs.PlateCarree())
ax.set_title(month_names[11])

#---------------------------------------------------------
```

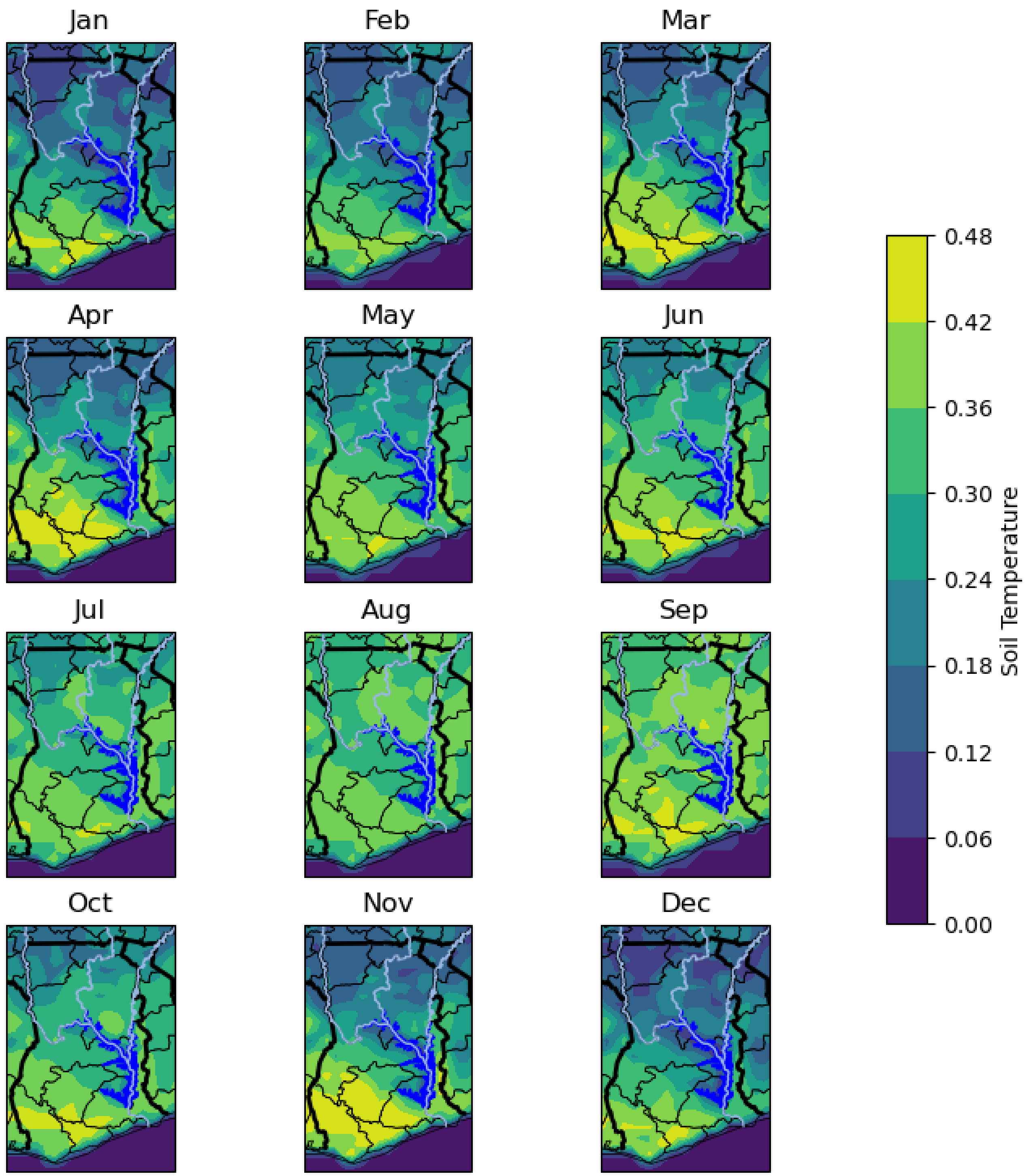
```
color_bar=fig.add_axes([0.82,0.29,0.025,0.5])
fig.colorbar(cb,cax=color_bar,label='Soil Temperature')
fig.subplots_adjust(wspace=-0.55, top=0.93);
```



In [173…

```
#alternatively
fig,ax=plt.subplots(4,3,figsize=(10,8.5),
                    subplot_kw={'projection': ccrs.PlateCarree()})
ax=ax.flatten()
month_names=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
for i in range(12):
    ax[i].add_feature(cfeature.COASTLINE.with_scale('110m'),linewidth=0.5)
    ax[i].add_feature(cfeature.BORDERS,linewidth=2)
    ax[i].add_feature(cfeature.STATES, linewidth=0.5)
    ax[i].add_feature(cfeature.OCEAN)
    ax[i].add_feature(cfeature.LAKES, color='blue')
    ax[i].add_feature(cfeature.RIVERS)
    ax[i].set_extent([-3.4,1.4,4.5,11.5])
    ax[i].set_title(month_names[i])
```

```
    cb= ax[i].contourf(swvl1_clim.longitude, swvl1_clim.latitude, swvl1_clim[i], transfo
    color_bar=fig.add_axes([0.82,0.29,0.025,0.5])
fig.colorbar(cb,cax=color_bar,label='Soil Temperature')
fig.subplots_adjust(wspace=-0.55, top=0.93);
```



In [ ]: