Scientific Data Analysis and Visualization with Python

Book · February 2023

DOI: 10.5281/zenodo.7592894

8 authors, including:

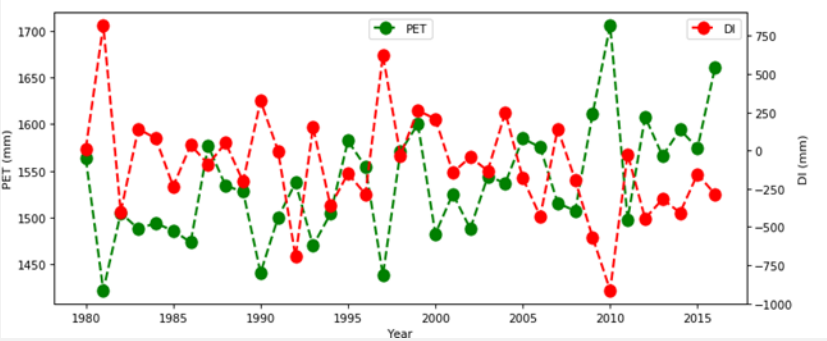
Md. Jalal Uddin
Zhejiang University
19 PUBLICATIONS **476** CITATIONS
SEE PROFILE

Nishat Rayhana Eshita
Jahangirnagar University
4 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Naiem Sheikh
Jahangirnagar University
4 PUBLICATIONS **1** CITATION
SEE PROFILE

Afifa Talukder
Jahangirnagar University
4 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Programming, Data Analysis, and Visualization with Python



Editors

Professor Dr Yubin Li

Professor Dr Golam Dastagir

Written by

Md. Jalal Uddin

Nishat Rayhana Eshita

Md. Asif Newaz

Naiem Sheikh

Afifa Talukder

Aysha Akter

Md. Habibur Rahman

Md. Babul Miah

https://researchsociety20.org/about-us/

ISBN: 978-984-35-3614-3

Codes including data are available at: https://doi.org/10.5281/zenodo.5944708

Author affiliations

Md. Jalal Uddin
Postdoctoral Researcher at Chonnam National University, South Korea

E-mail: founder-and-director@researchsociety20.org

Nishat Rayhana Eshita
Department of Environmental Sciences, Jahangirnagar University, Bangladesh

E-mail: nishateshita@gmail.com

Md. Asif Newaz
Research Consultant, Remote Sensing Division, Center for Environmental and Geographic Information Services (CEGIS), Bangladesh

E-mail: asifku24@gmail.com

Naiem Sheikh
Department of Geography and Environment, Jahangirnagar University, Bangladesh

E-mail: sheikh.46@geography-juniv edu.bd

Afifa Talukder
Department of Environmental Sciences, Jahangirnagar University, Bangladesh

E-mail: afifatalukdar@gmail.com

Aysha Akter
Environmental Science Discipline, Khulna University, Bangladesh

E-mail: ayshaku11@gmail.com

Md. Habibur Rahman
Institute of Disaster Management and Vulnerability Studies, University of Dhaka, Bangladesh

E-mail: habiburrahmanidmvs710@gmail.com

Md. Babul Miah
MS in Geo-information Science and Earth Observation, Patuakhali Science and Technology University (PSTU).

E-mail: babul.bsmrstu16@gmail.com

Table of Contents

1.1 Introduction to an arithmetic operator

Similar to any other programming language, arithmetic operators in python are used to perform mathematical operations such as addition, subtraction, etc.. Table 1.1 shows the application of these operators.

Table 1.1. Arithmetic operators in Python

Operator	Name	Example
+	Addition	a + b
-	Subtraction	a – b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b
**	Exponentiation	a ** b
//	Floor division	a // b

1.2 Operand

In Python, operators are special symbols that are used for computation. The values on which an operator acts are called operands. Here is an example.

```
a = 10
b = 20
```

```
a + b
30
```

In this case, the '+' operator adds the operands 'a and b' together. An operand can be either a value or a variable that references an object.

```
a = 10
b = 20
a + b - 5
25
```

A sequence of operands and operators (e.g., a + b - 5) is called an expression.

1.3 Variables and arithmetic operations

A variable is a way to store values. In other words, a name given to a storage area that any program can manipulate is called variable.

a) Define variable in Python

The creation of variables in Python is simple. We just need to write the name of the variable with the assigned value by using the equal sign ("="), as shown below.

```
a = 5
b = 6
```

Here, 'a and b' are variables with assigned values using equal signs. These variables will be used for the next calculation of operators.

b) Addition by Syntax

The syntax in Python means how program will be written and interpreted.

For the addition of two numbers in python, we will use the **"+"** operator and then the print function will be used to get the final output. We will define another new variable "c" with an assigned formula of addition for two variable values. The syntaxes are given below:

```
a = 5
```

```
b = 6
c = a + b
print('Addition by Syntax = ', c)
```

In Python, anything inside the quotes is a string. We can use either single quotes or double quotes to define strings that can be numeric or non-numeric data.

c) Addition by Function

In the previous section, we learned how to perform addition by the operator. Now, we will perform the same operation by using the function. This function is contained in a library named operator. For the addition of values, the operator has added a function. First, we need to import the operator. After importing the library, we will call the add function from the library by "operator.add()". The syntaxes are given below:

```
import operator
a = 5
b = 6
d = operator.add(a, b)

print('Addition by Function = ', d)
```

Here, the operator library has been imported. Variables a and b are assigned with values. Variable d is assigned for adding the values of a and b by "operator.add()" function.

d) Subtraction by Syntax

In Python, "–" is the subtraction operator. It is used to subtract the second value from the first value. The syntaxes of subtraction are given below:

```
a = 5
b = 6
subs = a-b

print('Subtraction by Syntax = ',subs)
```

Here, variable “subs” is assigned with value to subtract the variable b from the variable a.

e) Subtraction by Function

We will perform subtraction by the “operator.sub(a,b)” function. This function returns the difference between the given arguments.

```
import operator
a = 5
b = 6
subs_1 = operator.sub(a, b)

print('Subtraction by Function = ', subs_1)
```

Here, variable “subs_1” is assigned for calculating the difference between two values of variables by using “operator.sub(a,b)” function.

f) Multiplication by Syntax

In Python, “*” is the multiplication operator. It is used to find the product of two values of two variables.

Example:

```
a = 5
b = 6
mul = a*b

print('Multiplication by Syntax = ',mul)
```

Variable “mul” is assigned with the multiplication of the values of a and b variables by using asterix (*) sign.

g) Multiplication by Function

We can complete multiplication of two values using operator.mul () function from operator library instead of asterix (*) sign operator. This function returns product of the given arguments. Example:

```
a = 5
b = 6
mul_1 = operator.mul (a, b)

print('Multiplication by Function = ',mul_1)
```

Here, variables a and b are being multiplied in assigned variable mul_1 by using operator.mul () function.

Output:

```
Multiplication by Function =  30
```

h) Division by Syntax

The division operator in Python is "/". It is used to find the quotient when the first operand is divided by the second. An example of this operation is given below:

```
a = 5
b = 6
div = a/b
print('division by Syntax = ',div)
```

Here, variable div is assigned with the a/b. That means it will express the quotient of two variables a and b.

Output:

```
division by Syntax =  0.8333333333333334
```

i) Division by Function

The function "truediv()" is used to divide two values. This function returns the division of the given arguments. It will express the actual result of division.

Example:

```
a = 5
b = 6
div_1 = operator.truediv(a, b)

print('division by Function = ',div_1)
```

Output:

```
division by Function =  0.8333333333333334
```

j) Floor division by Syntax

The floor division in python is determined by "//". It is used to find the whole number of the quotient when the first operand is divided by the second.

Example 1:

```
a = 5
b = 6
fldiv = a//b

print('Floor division by Syntax  = ',fldiv)
```

Output:

```
Floor division by Syntax  =  0
```

Example 2:

```
p = 6/5
p
```

Output:

```
1.2
```

Example 3

```
p = 6//5
```

```
p
```

Output:

```
1
```

k) Floor division by Function

We can perform floor division by ‘operator.floordiv(a,b)’ function. This function also returns the division of the given arguments. But the value is floored value that returns the greatest small integer.

```
a = 5
b = 6
fldiv_1 = operator.floordiv(a, b)

print('Floor division by Function = ',fldiv_1)
```

Output:

```
Floor division by Function =  0
```

l) Modulus (remainder) by Syntax

The modulus operator in Python is “%”. When the first operand is divided by the second, it is used to calculate the remainder. The operands can be either integers or floats.

Example 1:

```
a = 5
b = 6
mod = a%b

print('Modulus by Syntax = ',mod)
```

Output:

```
Modulus by Syntax =  5
```

Example 2:

```
q = 6%5
q
```

Output:

```
1
```

m) Modulus (remainder) by Function

We can find out the remainder of a division between two operands by using the operator.mod(a,b) function. This function resides in the operator library.

```
a = 5
b = 6
mod_1 = operator.mod(a, b)

print('Modulus by Function = ',mod_1)
```

Output:

```
Modulus by Function =  5
```

n) Exponent (power) by Syntax

In Python, "**" is the exponentiation operator. It is used to raise the first operand to the power of the second.

```
a = 5
b = 6
exp = a**b              # a**b (5*5*5*5*5*5 = 15625)

print('Exponent by Syntax = ',exp)
```

Output:

```
Exponent by Syntax =  15625
```

o) Exponent (power) by Function

In python, operator.pow(a,b) returns the value of a to the power of b (a^b).

Example:

```
a = 5
b = 6
exp_1 = operator.pow(a, b)
print('Exponent by Function = ',exp_1)
```

Output:

```
Exponent by Function =  15625
```

p) All arithmetic operators by Syntax

Let's take a simple example to understand all the arithmetic operators by syntax in a python program by including all of them in a single program below:

```
a = 5
b = 6

c    = a + b
subs  = a-b
mul   = a*b
div   = a/b
fldiv = a//b
modu  = a%b
exp   = a**b
```

```
print('Addition by Syntax = ',c)
print('Subtraction by Syntax = ',subs)
print('Multiplication by Syntax = ',mul)
print('Division by Syntax = ',div)
print('Floor division by Syntax = ',fldiv)
print('Reminder by Syntax = ',modu)
print('Power or exponent by Syntax = ',exp)
```

Outputs:

```
Addition by Syntax =  11
Subtraction by Syntax =  -1
Multiplication by Syntax =  30
Division by Syntax =  0.8333333333333334
Floor division by Syntax =  0
Reminder by Syntax =  5
Power or exponent by Syntax =  15625
```

q) All arithmetic operators by Function

Let's take a simple example to understand all the arithmetic operators by function in a Python program by including all of them in a single program below:

```
import operator         # import library

# define variables

a = 5
b = 6

d = operator.add(a, b)
e = operator.sub(a, b)
f = operator.mul(a, b)
g = operator.truediv(a, b)
```

```
h = operator.floordiv(a, b)
i = operator.mod(a, b)
j = operator.pow(a, b)

print('Addition by Function = ',d)
print('Subtraction by Function = ',e)
print('Multiplication by Function = ',f)
print('Division by Function = ',g)
print('Floor division by Function = ',h)
print('Reminder by Function = ',i)
print('Power or exponent by Function = ',j)
```

Output:

```
Addition by Function =  11
Subtraction by Function =  -1
Multiplication by Function =  30
Division by Function =  0.8333333333333334
Floor division by Function =  0
Reminder by Function =  5
Power or exponent by Function =  15625
```

1.4 Practice with real data

We will calculate the average temperature from the minimum and maximum temperature

Equation: avg_temp = (min_temp + max_temp)/2

a) Import library and dataset

Pandas library

Pandas is one of the most popular and favourite library packages in Python for data file import and analysis. Pandas allow importing data from various file formats such as comma-separated values, JSON, SQL database tables or queries, Microsoft Excel, etc.

In pandas library, read_csv function is used for csv file and read_excel is used for excel files. Essential commands are given below:

Syntax

```
import pandas as pd

dataset = pd.read_csv('class-1_arithmetic_operators.csv')

dataset.head()
```

Here, pd is the short form of imported pandas' library. The variable dataset is defined with an imported CSV file which represents year, min_temp and max_temp. In Python, by default head() function represents data of the first five rows with all columns.

Output:

	Year	min_temp	max_temp
0	1980	11.810000	25.603226
1	1981	13.103333	24.832258
2	1982	12.441935	26.622581
3	1983	11.929032	25.203226
4	1984	11.670968	25.083871

If we want to know how many rows and column are being existed in the data sheet, we can use the shape() function. In Python, shape () function represents a number of all columns and rows.

```
dataset.shape
```

Output:

```
(38, 3)
```

After using the shape function, we can easily understand that thirty-eight rows and three columns are being existed in the data file. Now, we will define two variables from the data file for average temperature calculation.

```
min_tem = dataset.min_temp

max_tem = dataset.max_temp
```

b) Calculate average temperature by Syntax

For the calculation of average temperature, we will add the max_temp and min_temp. After that, the total addition will be divided by two. Finally, the average temperature will be expressed.

```
avg_temp = (min_tem + max_tem)/2


print('average temperature by Syntax = \n', avg_temp.head())
```

Here, "\n" indicates new line creation. The head function represents the first five rows of averaged data by default.

Output:

```
average temperature by Syntax =
 0    18.706613
1    18.967796
2    19.532258
3    18.566129
4    18.377419
dtype: float64
```

c) Calculate average temperature by Function

Now, we will calculate the average values of max_temp and min_temp by using add function from the operator library. Syntax is given below:

```
import operator        # import library


avg_temp_1 = operator.add(min_tem, max_tem)/2


print('average temperature by Function = \n',avg_temp_1.head())
```

Output:

```
average temperature by Function =
 0    18.706613
1    18.967796
2    19.532258
3    18.566129
4    18.377419
dtype: float64
```

1. Comparison operators

A comparison or relational operator in python compares two operands' values and returns True or False depending on whether the condition is met. We normally use this operators in decision-making. Less than, greater than, less than or equal to, greater than or equal to, equal to, and not equal to are examples of comparison operators (Table 2.1).

Table 2.1. Comparison operators in Python

Operator	Description	Syntax
>	Greater than	a > b
<	Less than	a < b
==	Equal to	a == b
!=	Not equal to	a != b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Now, let's see the examples of all the relational operators one by one for more understanding.

a) Greater than (>) operator

The greater than (>) an operator checks whether the value of the left operand is greater than the value of the right operand. The output of greater than is 'True' if the relation is correct otherwise the output will be False.

```
a = 5
b = 6
print('If a is greater than b =',a>b)
```

Output:

```
If a is greater than b = False
```

Here, the output of the print function is 'False' because the value of 'a' is less than the value of 'b', so the relation is wrong.

1.1. Less than (<) operator

The less than operator is used for checking if the value of the left operand is less than the value of the right operand. The output of < operator is 'True' if the relation is correct otherwise the output will be 'False'.

```
a = 5
b = 6
print('If a is less than b =',a<b)
```

Hence, the output will be 'True' because the value of 'a' is less than the value of 'b'. Therefore, the relation is correct.

Output:

```
If a is less than b = True
```

1.2. Equal to (==) operator

The equal to operator is used for checking if the value of two operands is equal. The output of equal to operator is 'True' if the relation is correct otherwise the output will be 'False'.

```
a = 5
b = 6
```

```
print('If a is equal to b =',a==b)
```

Output:

```
If a is equal to b = False
```

We observe that the output of the print function is 'False' because the values of the variable 'a' and 'b' are not the same, so the relationship is 'False'.

1.3. Not equal operator (!=)

In Python, not equal operator is denoted by "!=", and it is being used for checking if the value of the left operand is not equal to the value of the right operand. The output of the not equal operator is 'True' if this relation is correct otherwise the output will be 'False'.

Syntax

```
a = 5
b = 6
print('If a is not equal to b =',a!=b)
```

Output:

```
If a is not equal to b = True
```

In the above-mentioned example, the output of the print function is 'True' because the value of variable 'a' is not equal to the value of variable 'b'. Consequently, the relation is correct.

1.4. Greater than or equal to (>=) operator

Greater than or equal to operator in python is used for checking if the value of the left operand is greater than or equal to the value of the right operand. The output of '>=' operator is 'True' if this relation is correct otherwise the output will be 'False'.

```
a = 5
b = 6
```

```
print('If a is greater than or equal to b =',a>=b)
```

Output:

```
If a is greater than or equal to b = False
```

In this example, the output of print is ‘False’ because the value of variable ‘a’ is neither greater nor equal to the value of variable ‘b’. Thus, the relationship is ‘False’.

1.5. Less than or equal to the operator (<=)

In Python, less than or equal to operator is denoted by “<=” and used for checking if the value of the left operand is less than or equal to the value of the right operand. The output of this relational operator is ‘True’ if the mentioned relation is correct, otherwise, the output will be False.

```
a = 5
b = 6
print('If a is less than or equal to b =',a<=b)
```

Here, the output of the print function will be ‘True’ because the value of variable ‘a’ is not equal but less than the value of variable ‘b’. Consequently, the relation will be correct.

Output:

```
If a is less than or equal to b = True
```

1.6. All comparison or relational operators by Syntax

Let’s take a simple example to understand all the comparison or relational operators by syntax in a python program by including all of them in a single program below:

```
# Define variable
a = 5
b = 6
```

```
print('If a is greater than b =',a>b)

print('If a is less than b =',a<b)

print('If a is equal to b =',a==b)

print('If a is not equal to b =',a!=b)

print('If a is greater than or equal to b =',a>=b)

print('If a is less than or equal to b =',a<=b)
```

Output:

```
If a is greater than b = False
If a is less than b = True
If a is equal to b = False
If a is not equal to b = True
If a is greater than or equal to b = False
If a is less than or equal to b = True
```

1.7. All Comparison or relational operators by Function

Let's take simple examples which are previously mentioned to understand all the comparison or relational operators by operator function in Python program by including all of them in a single program below.

```
import operator as op   # Import library, op is short name

a = 5   # define a variable

b = 6

print('If a is greater than b =',op.gt(a,b))

print('If a is less than b =',op.lt(a,b))

print('If a is equal to b =',op.eq(a,b))

print('If a is not equal to b =',op.ne(a,b))

print('If a is greater than or equal to b =',op.ge(a,b))
```

```
print('If a is less than or equal to b =',op.le(a,b))
```

Output:

```
If a is greater than b = False
If a is less than b = True
If a is equal to b = False
If a is not equal to b = True
If a is greater than or equal to b = False
If a is less than or equal to b = True
```

We observe that the syntax operator and functional operator gave the same types of Boolean results for decision-making.

1.8. Example of comparison or relational operators for real data

We will monitor drought (water scarcity) for Rangpur station in Bangladesh from 1994 to 1995. The example has been adapted from Uddin et al. (2020). Source: https://link.springer.com/article/10.1007/s12517-020-05302-0

We have an excel file of data for analysis with a comparison operator for drought monitoring. Pandas is one of the most popular and favourite library packages in Python programming language for data file import and analysis. Pandas allow importing data from various file formats such as comma-separated values, JSON, SQL database tables or queries, Microsoft Excel, etc. To read the data file, we need to import the 'pandas' library first. In the pandas' library, the read_csv function is used for CSV files, and read_excel is used for excel files.

```
import pandas as pd

dataset = pd.read_excel('drought_data.xlsx')

dataset.head()
```

Here, the ‘pd’ is the short form of pandas library which is very helpful in data analysis. Variable “dataset” is assigned with data file using the ‘pandas’ library. Then, the ‘head()’ function will represent the first five rows of data with all columns.

Output:

	Year and month	SPI
0	1983-01	-0.285149
1	1983-02	-0.299666
2	1983-03	-0.229297
3	1983-04	-0.382996
4	1983-05	-0.060437

From the above-mentioned summary of data, we observe that data has two columns named “Year and month” and another one is “SPI”. The first five rows only showed in the output, which starts from zero because Python is zero-based programming language.

Indexing the desired data

Now, we want to open the specific column and rows for analysis from the original data file. For this reason, we need to create a new variable “SPI” is assigned with a specific name of column (SPI) from the data file by using the name of the data file variable. The slicing operator will be used for selecting specific rows. The implementation of the idea is given below.

```
SPI = dataset.SPI[133:156]   # 1994-1995


SPI.head()
```

Here, we perceive that variable SPI is assigned with SPI values with specific rows from the original data file. In addition, the ‘head ()’ function represents a snap of data. After running this program, the output will be:

```
133    0.289110
134    0.291538
135    0.238511
136    0.236418
137   -0.093880
Name: SPI, dtype: float64
```

Comparison or relational operators by Syntax and Function

Now, we will use comparison or relational operators by syntax and function for decision-making. For extreme drought, values of SPI must be less than or equal to 2. Implementation of this is given below:

```
# For extreme drought, SPI values must be <=-2
# Comparison by syntax
print('Extreme drought   = \n',SPI<=-2)

# Comparison by function
import operator as op
Threshold = -2
print('Extreme drought   = \n',op.le(SPI,Threshold))
```

Here, "\n" indicates one line feed. That means new line creation.

Output:

```
Extreme drought   =
 133    False
134    False
135    False
136    False
137    False
138    False
139    False
140     True
141     True
142     True
143     True
144     True
145     True
146     True
147     True
148     True
149     True
150     True
151    False
152    False
153    False
154    False
155    False
Name: SPI, dtype: bool
```

Now, we can easily interpret the locations where extreme drought is occurring.

Create a data frame of extreme drought

For monitoring extreme drought, we can create a data frame where only extreme drought time will have existed.

Syntax

```
extreme_drought = dataset[dataset['SPI']<=-2]


extreme_drought.head()
```

Output:

	Year and month	SPI
140	1994-09	-2.163714
141	1994-10	-2.230607
142	1994-11	-2.245852
143	1994-12	-2.300033
144	1995-01	-2.476514

This is the data frame of extreme drought time, where the SPI value is less than or equal to 2.

2. Logical operators

A logical expression is a statement, which can either be true or false. In the previous example, the mathematical expression a<b means that 'a' is less than 'b', and values of 'a' and 'b' where a≥b are not permitted. For more information, you can visit https://stackoverflow.com/questions/21415661/logical-operators-for-boolean-indexing-in-pandas.

Table 2.3. Logical operators in Python

Operator	Description	Syntax
and	Logical AND: If both the operands are true then the condition becomes true.	a and b
or	Logical OR: If any of the two operands are non-zero then the condition becomes true.	a or b
not	Logical NOT: Used to reverse the logical state of its operand.	Not a

Now, let's see the examples of all the logical operators one by one for more understanding.

```
a = 5
print("a and a is = ", a<10 and a>0)
```

```
print("a and a is = ", a>10 and a>0)

print("a or a is  = ", a>10 or a>0)

print("a or a is  = ", a>10 or a<0)

print("not a is   = ", not(a>10))
```

Output:

```
a and a is =  True
a and a is =  False
a or a is  =  True
a or a is  =  False
not a is   =  True
```

We can also use the numpy module for the logical_and operator.

```
import numpy as np   # library import

print("a and a is = ",np.logical_and(a<10,a>0))

print("a or a is  = ",np.logical_or(a>10, a>0))

print("not a is   = ",np.logical_not(a>10))
```

Output:

```
a and a is =  True
a or a is  =  True
not a is   =  True
```

2.7. Example of logical operator function for real data

Here, we used the previous example for ‘logical_and’ operator.

```
wet_climate = np.logical_and(SPI.gt(1), SPI.lt(2))

print("wet climate = \n", wet_climate)
```

Output:

```
wet climate =
 133    False
134    False
135    False
136    False
137    False
138    False
139    False
140    False
141    False
142    False
143    False
144    False
145    False
146    False
147    False
148    False
149    False
150    False
151    False
152    False
153    False
154    False
155    False
Name: SPI, dtype: bool
```

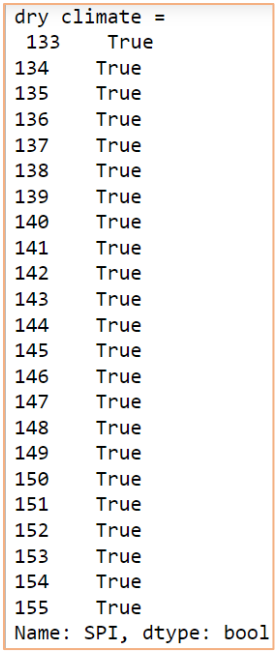
This output indicates the types of wet climates.

The values of SPI less than 1, less than or equal to -2 (SPI<1 or, SPI<=-2), indicates a dry climate.

```
dry_climate = np.logical_or(SPI.lt(1), SPI.le(-2))

print("dry climate = \n",dry_climate)
```

Output:



These results suggest that the weather type is dry.

For not extreme wet climates, the SPI value must be less than 2 (SPI<2).

```
print("not extreme wet climate = \n", np.logical_not(SPI>2))
```

Output:

```
not extreme wet climate =
 133    True
134    True
135    True
136    True
137    True
138    True
139    True
140    True
141    True
142    True
143    True
144    True
145    True
146    True
147    True
148    True
149    True
150    True
151    True
152    True
153    True
154    True
155    True
Name: SPI, dtype: bool
```

For monitoring wet climate (1<SPI>2), we can create a data frame where only wet climate time will have existed.

```
df_wet_climate = dataset[np.logical_and(dataset['SPI'].gt(1), dataset['SPI'].lt(2))]

df_wet_climate.head()
```

Output:

	Year and month	SPI
17	1984-06	1.477008
18	1984-07	1.814248
19	1984-08	1.753462
29	1985-06	1.541693
30	1985-07	1.540050

For monitoring dry climate (1>SPI>=-2), we can create a data frame where only dry climate time will have existed.

```
df_dry_climate = dataset[np.logical_and(dataset['SPI'].lt(1), dataset['SPI'].le(-2))]

df_dry_climate.head()
```

Output:

	Year and month	SPI
140	1994-09	-2.163714
141	1994-10	-2.230607
142	1994-11	-2.245852
143	1994-12	-2.300033
144	1995-01	-2.476514

1. Bitwise operators

In Python, Bitwise operators work only on integers and format of binary- prefix 0b included.

Python's "and", "or" and "not" logical operators are designed to work with "scalars". However, "&", "|" and "~" Bitwise operators are designed to work with both "scalars" and "vectors".

Table 3.1. Bitwise operators in Python

Operator	Description	Syntax
&	Bitwise AND: Sets each bit to 1 if both bits are 1	a & b
|	Bitwise OR: Sets each bit to 1 if one of the two bits is 1	a | b
~	Bitwise NOT: Inverts all the bits	~a
^	Bitwise XOR: Sets each bit to 1 if only one of two bits is 1	a ^ b
>>	Bitwise right shift: Shift right by pushing copies of the leftmost bit in from the left, and letting the rightmost bits fall off	a>>
<<	Bitwise left shift: Shift left by pushing zeros in from the right and letting the leftmost bits fall off	a<<

For more information, you may visit at: https://www.geeksforgeeks.org/python-bitwise-operators/.

```
a = 4

b = 12
```

Print the variables into binary by using ‘bin()’ function.

```
print("decimal to binary for a =",bin(a))

print("decimal to binary for b =",bin(b))
```

Output:

```
decimal to binary for a = 0b100
decimal to binary for b = 0b1100
```

```
print("a & b =", a&b)

print("a | b =", a | b)

print("~a =", ~a)
```

Output:

```
a & b = 4
a | b = 12
~a = -5
```

```
import operator as op

print("a & b =", op.__and__(a, b))

print("a | b =", op.__or__(a, b))
```

Output:

```
a & b = 4
a | b = 12
```

Example of Bitwise operators for real data

Here, we used previous example for Bitwise operator.

```
import pandas as pd

import numpy as np

dataset = pd.read_excel('drought_data.xlsx')

dataset.head()
```

Output:

	Year and month	SPI
0	1983-01	-0.285149
1	1983-02	-0.299666
2	1983-03	-0.229297
3	1983-04	-0.382996
4	1983-05	-0.060437

```
SPI = dataset.SPI[133:156]   # 1994-1995

SPI.head()
```

```
133    0.289110
134    0.291538
135    0.238511
136    0.236418
137   -0.093880
Name: SPI, dtype: float64
```

```
print("wet climate = \n",SPI.gt(1) & SPI.lt(2))
```

Output:

```
wet climate =
 133    False
134    False
135    False
136    False
137    False
138    False
139    False
140    False
141    False
142    False
143    False
144    False
145    False
146    False
147    False
148    False
149    False
150    False
151    False
152    False
153    False
154    False
155    False
Name: SPI, dtype: bool
```

```
print("dry climate = \n",SPI.lt(1) | SPI.le(-2))
```

Output:

```
dry climate =
 133    True
134    True
135    True
136    True
137    True
138    True
139    True
140    True
141    True
142    True
143    True
144    True
145    True
146    True
147    True
148    True
149    True
150    True
151    True
152    True
153    True
154    True
155    True
Name: SPI, dtype: bool
```

```
print("not extreme wet climate = \n", ~(SPI>2))
```

Output:

```
not extreme wet climate
 133    True
134    True
135    True
136    True
137    True
138    True
139    True
140    True
141    True
142    True
143    True
144    True
145    True
146    True
147    True
148    True
149    True
150    True
151    True
152    True
153    True
154    True
155    True
Name: SPI, dtype: bool
```

```
print("wet climate = \n",op.__and__(SPI.gt(1), SPI.lt(2)))
```

Output:

```
wet climate =
 133    False
134    False
135    False
136    False
137    False
138    False
139    False
140    False
141    False
142    False
143    False
144    False
145    False
146    False
147    False
148    False
149    False
150    False
151    False
152    False
153    False
154    False
155    False
Name: SPI, dtype: bool
```

```
print("dry climate = \n",op.__or__(SPI.lt(1), SPI.le(-2)))
```

Output:

```
dry climate =
 133    True
134    True
135    True
136    True
137    True
138    True
139    True
140    True
141    True
142    True
143    True
144    True
145    True
146    True
147    True
148    True
149    True
150    True
151    True
152    True
153    True
154    True
155    True
Name: SPI, dtype: bool
```

2. Assignment and special operators (identity and membership) in Python

The most common assignment operator is the equal sign "=".For example, a = 15 assigns the value of the integer 15 to the variable a.

a) Assignment operators

```
a = 15
b = 10
c = 5
d = 3
e = 4
f = 15
g = 7

a += 10                 # a + 10
print ("a =",a)

b *= 10                 # b * 10
print ("b =",b)

c /= 10                 # c / 10
print ("c =",c)

d %= 10                  # d % 10
print ("d =",d)

e **= 10                 # e ** 10
print ("e =",e)

f //= 10                # f // 10
print ("f =",f)

g -= 10                  # g - 10
```

```
print ("g =",g)
```

Output:

```
a = 25
b = 100
c = 0.5
d = 3
e = 1048576
f = 1
g = -3
```

b) Special operators

Python language offers some special types of operators like the identity operator or the membership operator.

Identity operators check if two values or variables are located on the same part of the memory. While membership operators test whether a value or variable is found in a sequence. For more information, you may visit at https://www.programiz.com/python-programming/operators.

Syntax of Identity operators: "is" and "is not"

```
a = ["Jalal", "Ripa"]
b = ["Jalal", "Ripa"]
print(a is b)      # Though a and b have the same content, a is not the same object as b.

c = 'Jalal'
d = 'Jalal'
print(c is d)  # returns True because c and d are located on the same part of the memory.

print(a is not b)
```

Output:

```
False
False
True
```

Membership operators ("in" and "not in")

In Python, ‘in’ and ‘not in’ are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

Syntax

```
a = ["Jalal", "Ripa"]
b = ["Jalal", "Ripa"]

print("Jalal" in a)
print("Jalal" not in a)
```

Output:

```
True
False
```

Syntax

```
male   = "Jalal"
female = "Ripa"

name_list = ["Jalal","Ripa","Jara","Zahura"]

if (male not in name_list):
   print("male is NOT exist in name_list")
else:
   print("male is exist in name_list")
```

Output:

```
male is exist in name_list
```

Syntax

```
if (female in name_list):
   print("female is exist in name_list")
else:
```

```
print("female is NOT exist in name_list")
```

Output:

```
female is exist in name_list
```

Chapter 4: Application of input, range, string, index operator, slicing operator, concatenation, and delete in Python

1. Use of "input()" function

The 'input()' function in python is used to get input from the user. For example, it can be a name, number or full sentence. After running the input function, it prompts the screen and asks for input. After reading the input, it converts it into a string even if the input is a number. The codes are given below:

```
x = input("Enter your name: ")   # The input() method converts into a string and returns it.

print("Hello, " + x)

print(type(x))
```

Output:

```
Enter your name: Naiem
Hello, Naiem
<class 'str'>
```

However, it is also possible to get an integer output from 'input()' function. To get an integer output, we need to modify the code. The code is given below:

```
y = int(input('enter age: '))  # int = integer

print(type(y))
```

Output:

```
enter age: 25
<class 'int'>
```

2. String in python

The string data type in python is a sequence of characters. These characters are different symbols. It may include alphabets, numbers or anything, but the condition is the string must be written in quotation marks. For example ‘Hello’, “Hello Jalal”, ““Hello Jalal””

Codes are given below:

```
str_ = 'Hello Jalal'

print(str_)

str1 = "Hello Jalal"

print(str1)

str2 = """Hello Jalal"""

print(str2)
```

Output:

```
Hello Jalal
Hello Jalal
Hello Jalal
```

3. Indexing

The indexing in python is a way of calling a particular element by its position. By indexing, one can access any element of choice directly. Python starts counting from zero, so the first element is indexed as 0 then 1, 2 and so on.

In the previous example, the “Hello Jalal” indexing number is as follows:

```
0 1 2 3 4  5 6 7 8 9

H e l l o  j a l a l
```

Codes for indexing are given below:

```
print(str_[0])      # returns the first item

print(str_[1])      # returns the second item
```

Reverse indexing is also used in python, which starts from -1. The '-1' stands for the last element of a series. For the above-mentioned example, reverse indexing is given below:

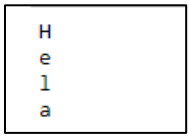
```
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1

H  e  l  l  o  J  a  l  a  l
```

```
print(str_[-1])      # returns the first item from the end

print(str_[-2])      # returns the second item from the end
```

Output:



4. Slicing Operator

The slicing operator is used to call a certain slice of a data set. "**:**" is used for slicing. Codes for slicing are given below:

```
print(str_[0:3])
```

Output:

```
Hel
```

5. Concatenation

Concatenation is a way of adding two or more different strings together. The "+" operator is used to connect strings. An example is given below:

```
con1 = str_ + str1

print(con1)
```

Output:

```
Hello JalalHello Jalal
```

6. Repeat the String

To print a string multiple times, the “*” operator is used. If we want to repeat a string thrice, the codes will be the following:

```
print(str_*3)
```

Output:

```
Hello JalalHello JalalHello Jalal
```

7. Convert string to upper case

To convert a string into upper case, the ‘.upper()’ function can be used. For example,

```
"Jalal".upper()
```

Output:

```
'JALAL'
```

8. Converting String to lowercase

To convert a string into lower case, the ‘.lower()’ function can be used. For example,

```
"Jalal".lower()
```

Output:

```
'jalal'
```

9. Capitalizing the first letter of every word

To capitalize the first letter of every word, the ‘.title()’ functionis used. For Example,

```
"hello jalal".title()
```

Output:

```
'Hello Jalal'
```

10. Replacing String

To replace one string with another, the '.replace()' function is used herein.

```
"hello jalal".replace('h', 'H')
```

Output:

```
'Hello jalal'
```

11. Split String

To split the string, the '.split()' function can be used. For example,

```
"hello jalal".split()
```

Output:

```
['hello', 'jalal']
```

12. Join String

To join strings, the following codes are used:

```
' '.join(['hello', 'jalal'])
```

Output:

```
'hello jalal'
```

13. Delete String

In our previous examples, we have used a string, which is assigned in a variable named str_ 3. Now, we want to delete the string. Codes are given below.

```
str_3 = "Hello Jalal"
print(str_3)
Hello Jalal
del str_3
```

The 'del str_3' removes the variable 'str_3'. Now, if we want to print str_3, there will show an error.

```
print(str_3)
```

Output:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-9e3ec027d66c> in <module>
----> 1 del str_3
      2
      3 print(str_3)

NameError: name 'str_3' is not defined
```

14. Range

To create a range (start, stop, and step), a sequence of numbers is used.

Start refers to the number from where the counting starts, stop refers to the number where the sequence ends and step refers to the difference between numbers. Codes are given below:

```
range_of_data = range(1,10)

print(range_of_data)
```

Output:

```
range(1, 10)
```

1. Dictionary in Python

In Python, dictionaries are used to store data. It is an unordered collection of items. There are certain features of a dictionary. They are given below:

i) Dictionaries are written in a pair of curly brackets.
ii) Each item has keys and values.
iii) Items are unordered.
iv) Items are changeable.
v) Dictionaries do not allow duplicate items.

Example of a dictionary:

```
mydict = {"name": "Jalal","wife": "Ripa Jalal","kids":"Jara Jalal","year of birth": 1990,"good man":True,"favourite colors": ["white", "green", "red"]}

print(mydict)
```

Output:

```
{'name': 'Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['white', 'green', 'red']}
```

1.1. Creating a dictionary using ‘dict()’ function

Another way of creating a dictionary is by using the ‘dict()’ function. If no arguments are passed in the ‘dict()’ function, it will create an empty dictionary.

Example:

```
dict_with_funt = dict(name = "Jalal", wife = "Ripa Jalal", kids = "Jara Jalal", year_of_birth = "1990", good_man = True, favourite_colors = ["white", "green", "red"])

print(dict_with_funt)
```

Output:

```
{'name': 'Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year_of_birth': '1990', 'good_man': True, 'favourite_colors': ['white', 'green', 'red']}
```

1.2. Nested dictionary

A nested dictionary refers to a dictionary, which includes another dictionary. It is a dictionary within a dictionary which already exists.

An example can make the concept clearer:

```
training = {
  "Research Society" : {
    "course name" : "Python",
    "participants level":"all",
    "year" : 2020
  },

  "IMSA" : {
    "course name" : "Matlab",
    "participants level":"all",
    "year" : 2020
  },

  "CUSS and RS" : {
    "course name" : "Statistics",
    "participants level":"all",
    "year" : 2020
```

```
  }

}



print(training)
```

Here, the dictionary “training” includes another 3 dictionaries “Research Society”, “IMSA”, and “CUSS and RS”.

Output:

```
{'Research Society': {'course name': 'Python', 'participants level': 'all', 'year': 2020}, 'IMSA': {'course name': 'Matlab', 'p
articipants level': 'all', 'year': 2020}, 'CUSS and RS': {'course name': 'Statistics', 'participants level': 'all', 'year': 202
0}}
```

1.3. Merge dictionary

Merge dictionary means joining multiple dictionaries together and generating a new dictionary. Let, there are three dictionaries named Research_Society, IMSA and CUSS_RS that we want to merge.

Example:

```
Research_Society = {

    "course name" : "Python",

    "participants level":"all",

    "year" : 2020

}



IMSA = {

    "course name" : "Matlab",

    "participants level":"all",
```

```
    "year" : 2020
}

CUSS_RS = {
    "course name" : "Statistics",
    "participants level":"all",
    "year" : 2020
}

training = {
  "Research Society" : Research_Society,
  "IMSA" : IMSA,
  "CUSS and RS" : CUSS_RS
}

print(training)
```

Output:

```
{'Research Society': {'course name': 'Python', 'participants level': 'all', 'year': 2020}, 'IMSA': {'course name': 'Matlab', 'participants level': 'all', 'year': 2020}, 'CUSS and RS': {'course name': 'Statistics', 'participants level': 'all', 'year': 2020}}
```

1.4. Duplicates are not allowed in dictionaries

In previous sections, we came to know that dictionaries do not allow duplicate items. Now, we will see an example of this. Let, in a list, we have two elements both of them are having the same key named “Year”. Now, let’s try to print the dictionary. The necessary codes are given below:

```
RS = {
   "course name" : "Python",
   "participants level":"all",
   "year" : 2019,
   "year" : 2020
}
print(Research_Society)                # it is not allowed to have two items with the same key
                                                    # overwrite existing values
```

Output:

```
{'course name': 'Python', 'participants level': 'all', 'year': 2020}
```

In the output, we can see that year 2019 is omitted, only the year 2020 is printed.

1.5. Print the data type of a dictionary

To print the data type of a dictionary, the following codes are used:

```
print(type(dict_with_funt))                # dictionaries are defined as objects with the data type
```

Output:

```
'dict'

<class 'dict'>
```

1.6. Print an element from the dictionary

To print any element from the dictionary, we need to write the following codes along with the key of the element. Let we want to print the name from the dictionary '**dict_with_funt'.**

Example:

```
print(dict_with_funt["name"])
```

Output:

```
Jalal
```

1.7. Print the number of items in the dictionary

To print the number of total elements of a dictionary, the following commands are used:

```
print(len(dict_with_funt))
```

Output:

```
6
```

1.8. Copy the dictionary

There are two ways of coping with a dictionary. One is by using '.copy()' function, and another is by using syntax. Let, the 'mydict' is a previously assigned dictionary that we want to copy.

Example with '.copy()' function:

```
new_dict = mydict.copy()                    # make a copy of a dictionary

print(new_dict)
```

Output:

```
{'name': 'Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['wh
ite', 'green', 'red']}
```

Example with the syntax:

```
new_dict1 = dict(new_dict)  # make a copy of a dictionary

print(new_dict1)
```

Output:

```
{'name': 'Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['wh
ite', 'green', 'red']}
```

1.9. Update dictionary

To change the value of any key, we can write the following codes. Let we want to change the value of the key “name”. Previously the value of the key name was “jalal” and now we want to replace it with “jara jalal”.

```
new_dict.update({"name": "Jara Jalal"})    # update value

print(new_dict)
```

Output:

```
{'name': 'Jara Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['white', 'green', 'red']}
```

1.10. Insert an item in the dictionary

To insert any new item in the dictionary, we need to write the following codes:

```
new_dict.update({"trainers": ["RS", "IMSA", "CUSS"]})

print(new_dict)
```

Output:

```
{'name': 'Jara Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['white', 'green', 'red'], 'trainers': ['RS', 'IMSA', 'CUSS']}
```

1.11. Get the value of any key

To get the value of any key, the following codes are needed:

```
get_value = new_dict.get("name”)

print(get_value)
```

Output:

```
Jara Jalal
```

1.12. Return the dictionary's key-value pairs

To get all the key-value pairs, the following codes are required.

```
items = new_dict.items()
```

```
print(items)
```

Output:

```
dict_items([('name', 'Jara Jalal'), ('wife', 'Ripa Jalal'), ('kids', 'Jara Jalal'), ('year of birth', 1990), ('good man', Tru
e), ('favourite colors', ['white', 'green', 'red']), ('trainers', ['RS', 'IMSA', 'CUSS'])])
```

1.13. Return key only

Codes to get the key only:

```
keys = new_dict.keys()

print(keys)
```

Output:

```
dict_keys(['name', 'wife', 'kids', 'year of birth', 'good man', 'favourite colors', 'trainers'])
```

1.14. Returning ‘value’ only

To get the values only, we need the following codes:

```
values = new_dict.values()

print(values)
```

Output:

```
dict_values(['Jara Jalal', 'Ripa Jalal', 'Jara Jalal', 1990, True, ['white', 'green', 'red'], ['RS', 'IMSA', 'CUSS']])
```

1.15. Remove a specific item from the dictionary

The following codes show how to remove the item from the dictionary.

```
new_dict.pop("name")

print(new_dict)
```

Output:

```
{'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['white', 'green', 'r
ed'], 'trainers': ['RS', 'IMSA', 'CUSS']}
```

1.16. Removing the last elements from the dictionary

The following codes are used to remove the last elements from the dictionary.

```
new_dict.popitem()
print(new_dict)
```

Output:

```
{'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['white', 'green', 'r
ed']}
```

1.17. Removing all items from the dictionary

To remove all the items from the dictionary, the following codes are required:

```
new_dict.clear()

print(new_dict)
```

Output:

```
{}
```

2. List in Python

A list in Python is a special data structure with a changeable and ordered sequence of elements. The list is written in square brackets and items are separated by a comma.

Example:

```
list_item  = ['jalal','Ripa', 2, 4, 6]

print( list_item)
```

Output:

```
['jalal','Ripa', 2, 4, 6]
```

2.1. Indexing List Items

Indexing in Python starts with zero (0). Indexing refers to the position of an element in the list. Example:

```
 0    1    2  3  4

['jalal','Ripa', 2, 4, 6]
```

Here, the index of ‘jalal’ is 0, the index of ‘Ripa’ is 1, the index of 2 is 2, the index of 4 is 3, and the index of 6 is 4. Reverse indexing is also applicable for the list items.

2.2. Print any item on a list

To call any item on the list, we need to know the indexing number. Let, we want to print “Jalal”, “Ripa”, 4, and 6.

The codes are the following:

```
print(list_item[0])                    # returns the first item

print(list_item[1])                    # returns the second item

print(list_item[-1])                   # returns the first item from the end

print(list_item[-2])                   # returns the second item from the end
```

Output:

```
jalal
Ripa
6
4
```

2.3. Append item

To add an item to the list, the ‘append()’ function is used. The ‘append()’ function adds the new item as the last item of the list. An example is given below:

```
list_item.append(1)          # append() method appends an element to the end of the list

print(list_item)
```

Output:

```
['jalal','Ripa', 2, 4, 6,1]
```

Here, in ‘list_item’ list, there were [‘jalal’, ‘Ripa’, 2, 4, 6] elements. After using ‘.append(1)’, it is showing ['jalal','Ripa', 2, 4, 6,1].

2.4. Insert item

To insert any item at a desired position, the ‘insert()’ function is used. We need to mention the index of the position where we want to assign the new item.

Let, we want to include a new item “Tamim” at index 2. The codes are the following:

```
list_item.insert(2,'Tamim')          # insert() method inserts the specified value at the specified position

print(list_item)
```

Output:

```
['jalal','Ripa', 'Tamim', 2, 4, 6, 1]
```

A new item "Tamim" is added to the list.

2.5. Removing an item from the list

To remove any item from the list, we need the following command. Let we want to delete the item "Tamim" from the list.

```
list_item.remove('Tamim')

print(list_item)
```

Output:

```
['jalal','Ripa', 2, 4, 6, 1]
```

The item "Tamim" is removed.

2.6. Removing the last item

The following codes remove only the last item of the list.

```
list_item.pop()

print(list_item)
```

Output:

```
['jalal','Ripa', 2, 4, 6]
```

The last item on the list is deleted.

2.7. Delete all items

To remove all the items from the list, we need the following codes:

```
list_item.clear()

print(list_item)
```

Output:

```
[]
```

2.8. Sorting of list items

If we want to sort the items on a list, the following codes are required. Let a list is assigned as a “number” whose elements are [1,3,2,5,4]. To sort the numbers, we will use the function ‘.sort()’.

```
numbers = [1,3,2,5,4]

numbers.sort()

print(numbers)
```

Output:

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

2.9. Reverse the list

To reverse all the elements of the list, the ‘reverse()’ function is used. Let a function “number” has numeric elements which we want to reverse. The codes are the following:

```
numbers.reverse()

print(numbers)
```

Output:

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

2.10. Making a copy of the list

To make a copy of an existing list, the following codes are required. The ‘copy()’ function is used in this case.

```
new_number = numbers.copy()

print(new_number)
```

Output:

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

3. Set in Python

Set is used to store multiple items in a single variable. It is created with a special built-in function ‘set()’. The set is written in a curly bracket. Items are unordered and not indexed. Items are unchangeable, and no duplicate elements are allowed. A set cannot have a list or dictionaries.

Example:

```
myset = {"Name","Jalal","year of birth", 1990, "good man",True,"favourite_colors", ("white", "green", "red")}

print("myset =", myset)
```

Output:

```
myset = {True, 1990, 'year of birth', 'favourite_colors', ('white', 'green', 'red'), 'Name', 'Jalal', 'good man'}
```

From the output, we can see that the items in the set are not in a similar order as the input. This is why sets are unordered and do not have any index.

3.1. Create a set by ‘set()’ function

The ‘set()’ is a built-in function in Python. We can easily create a set using this function. An example is given below:

```
myset = set({"Name","Jalal","year of birth", 1990, "good man",True,"favourite_colors", ("white", "green", "red")})

print("myset =", myset)
```

Output:

```
myset = {True, 1990, 'year of birth', 'favourite_colors', ('white', 'green', 'red'), 'Name', 'Jalal', 'good man'}
```

3.2. Duplicates are not allowed

Sets do not allow duplicate numbers. If we input and duplicate a number in the set, Python will automatically delete one.

Example:

```
myset = {"Jalal","Ripa","Jalal"}

print(myset)
```

Output:

{“Ripa” , “Jalal”}

3.3. Join set

Let two sets myset1 and myset2 contain the following items. The ‘union()’ function joins two sets together.

```
myset1 = {"Name","Jalal","year of birth"}

myset2 = {1990, "good man","favourite_colors"}

myset3 = myset1.union(myset2)

print("myset =", myset3)
```

Output:

```
myset = {'year of birth', 'favourite_colors', 1990, 'Name', 'Jalal', 'good man'}
```

3.4. Add items

To add items to a set, the ‘update()’ function is used. Let we want to add an element myset2 in myset1. The codes are the following:

```
myset1.update(myset2)

print("myset =", myset1)
```

Output:

```
myset = {'year of birth', 'favourite_colors', 1990, 'Name', 'Jalal', 'good man'}
```

[N.B. Both ‘union()’ and ‘update()’ functions exclude duplicate numbers.]

3.5. Intersection () function

The ‘intersection ()’ function does the same thing as in mathematics intersection dose with sets. It also joins two sets but keeps the items which are common in both sets:

Example:

```
myset3 = {"Jalal","Ripa","year of birth"}

myset4 = {"Jalal", "Ripa","favourite_colors"}
```

```
myset5 = myset3.intersection(myset4)

print("myset =", myset5)
```

Output:

```
myset = {'Ripa', 'Jalal'}
```

3.6. Add new items using ‘add()’ command

By using add function, one can easily add any item in a set. Let we want to add a new element “Jara” to our list. The codes are the following:

```
myset3 = {"Jalal","Ripa","year of birth"}

myset3.add("Jara")

print("myset = ",myset3)
```

Output:

```
myset =  {'Ripa', 'year of birth', 'Jalal', 'Jara'}
```

3.7. Remove set items

Both ‘remove()’ and ‘discard()’ functions are used to remove the item from the set. An example is given below:

```
myset3 = {"Jalal","Ripa","year of birth"}

myset3.remove("Jalal")

print("remove Jalal = ",myset3)
```

Output:

```
{"Ripa",  "year of birth"}
```

```
myset3.discard("Ripa")

print(myset3)
```

Output:

{“year of birth’}

Another function ‘pop()’ is also used to remove the item. The ‘pop()’ function removes the last item from the list but as the set is not ordered, we cannot say which element will be deleted after using ‘pop()’ function.

```
myset4 = {"Jalal","Ripa","year of birth","Jimi"}

myset4.pop()

print("remove the last item = ",myset4)
```

Output:

```
remove the last item =  {'year of birth', 'Jalal', 'Jimi'}
```

3.8. Keep all items except duplicates by symmetric difference method

We have two sets, myset3 and myset4. They have the following items:

```
myset3 = {"Jalal","Ripa","year of birth"}

myset4 = {"Jalal", "Ripa","favourite_colors"}
```

symmetric_difference() function returns a new set which contains all the elements except those elements that are common in both myset3 and myset4. The codes are the following:

```
myset5 = myset3.intersection(myset4)

    print("myset =", myset5)
```

Output:

Myset = {“Ripa” , “Jalal”}

From the output, we can see that myset5 only contains those elements which were common in both myset3 and myset4.

3.9. Clear data

To clear the set completely, the ‘.clear()’ function is used. The codes are following

```
myset4 = {"Jalal","Ripa","year of birth"}

myset4.clear()

print("removes all the elements = ",myset4)
```

Output:

```
removes all the elements =  set()
```

3.10. Delete the set completely

To delete the set completely, the following codes are required.

```
myset4 = {"Jalal","Ripa","year of birth"}

del myset4

print("delete the set = ",myset4)
```

Output:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-b5b0310a1888> in <module>
      2
      3 del myset4                                  # delete the set completely
----> 4 print("delete the set = ",myset4)

NameError: name 'myset4' is not defined
```

4. Tuple in Python

A tuple is also used to store multiple items in a single variable where the items are unchangeable and ordered. Tuples are created in the first bracket, and items are separated by a comma.

Example:

```
tuple_   = ('Hello', 'Jalal')

print(tuple_)
```

Output:

```
('Hello', 'Jalal')
```

4.1. Example of mixed data type tuple:

```
tuple3  = ('Jalal','Ripa', 2, 4, 6)
```

```
print(tuple3)
```

Output:

```
('Jalal','Ripa', 2, 4, 6)
```

4.2. Example of a nested tuple:

```
tuple4  = ('Jalal','Ripa', [2, 4, 6])   # nested tuple

print(tuple4)
```

Output:

```
('Jalal','Ripa', [2, 4, 6])
```

4.3. Indexing operator

Indexing is as same as discussed in previous data types. The codes for indexing tuples are given below:

```
print(tuple1[0])    # returns the first item

print(tuple1[1])    # returns the second item

print(tuple1[-1])   # returns the first item from the end

print(tuple1[-2])   # returns the second item from the end
```

Output:

```
Hello
Jalal
Jalal
Hello
```

4.4. Slicing operator in Python

The slicing operator returns a specific part of data. The '**:**' is used for slicing. The index number is required for slicing. Required codes are given below:

```
Tuple3 = ('Jalal','Ripa', 2, 4, 6)

print(tuple3[0:2])
```

Output:

```
('Jalal','Ripa', 2)
```

4.5. Concatenation

Concatenation is used to join multiple strings together. The '+' is used for concatenation. Codes are given below:

Let, two tuples tuple_ and tuple2 which we want to join.

```
tuple_  = 'Hello', 'Jalal'

tuple2  = (1,2,3,4)

con = tuple_ + tuple2

print(con)
```

Output:

```
('Hello', 'Jalal', 1, 2, 3, 4)
```

4.6. Repeat the tuple

To repeat the tuple multiple times, the '*' sign is used. An example is given below:

```
print(tuple_*3)
```

Output:

```
('Hello', 'Jalal', 'Hello', 'Jalal', 'Hello', 'Jalal')
```

4.7. Delete

We cannot delete items from the list rather we can delete the entire tuple. The 'del' function is used to delete any tuple.

Codes are given below:

```
del tuple_

print(tuple_)
```

Output:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-a00055b7fc21> in <module>
      1 del tuple_
      2
----> 3 print(tuple_)

NameError: name 'tuple_' is not defined
```

4.8. Length of tuple

To know the length of the tuple, the ‘len()’ function is used. The codes are the following:

```
tuple5  = ('Hello', 'Jalal','1','Jalal','3')

print(len(tuple5))
```

Output:

```
5
```

It means tuple5 has 5 elements.

4.9. The ‘count()’ and ‘index()’ function

The ‘count()’ function is used to count the index of an element. In the function, we need to pass the item of interest. The ‘index()’ function is used to know the element of a certain index. Let we want to know the element which has the index 3. The codes are the following:

```
print(tuple5.count('Jalal'))
```

Output:

```
2
```

```
print(tuple5.index('3'))
```

Output:

```
4
```

Chapter 6: Looping (for, while) and conditional statement (if, else) in Python

1. Loop in Python

Sometimes we need to do the same types of work again and again. This repetition of work is more time-consuming and harder. Normally, we do not like this type of work. However, Python programming language has a built-in function named “loop”. With the help of “loop”, we can conduct the same types of work again and again. This is not time-consuming and harder. There are two ways to create loops in Python: with the for-loop and the while-loop.

1.1. For Loop

When we have a block of code that needs to be repeated a certain number of times, we may use ‘for’ loops. It is used for iterating over an iterable object like a list, string, tuple, etc. The ‘for’ statement in Python iterates through the members of a sequence, executing the block each time. The implementation of an idea is given:

```
for i in range(20):        # start = 0, end = n-1
    print(i)
```

Here, we used ‘range()’ function that returns an immutable sequence of numbers between the given range. It is used when a user has to do an operation a particular number of times. Therefore, to generate a sequence of numbers, we can use the ‘range()’ function.
We can also define the start, stop and step size of ‘range(start, stop, step)’ function:

- start: Optional. An integer starting from which the sequence of integers is to be returned. This is an inclusive part of the sequence.
- stop: Required. An integer before which the sequence of integers is to be returned.
  The range of integers ends at n–1. This is an exclusive part of the sequence.
- step: Optional. An integer value determines the increment between each integer in the sequence. Its default value is 1 if not provided.

Example 1:

The ‘range(20)’ function will generate 0 to 19 and ‘for loop’ print each number from this sequence individually. In addition, on each iteration, the variable “i” takes the value of the item inside the sequence.

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

From this output, we observe that the iteration of numbers starts from 0 and ends at 19 because Python is 0 based and the ends point will be always n-1. However, the total count number of series is 20.

Example 2:

If we want to print 1 to 20 that means iteration starts from 1, ends at 19 and step size by default 1, so we write that:

```
for j in range(1,20):
    print(j)
```

Here, 1 is the starting point of the sequence of the number mentioned in the range function. Therefore, the range function generates a sequence of numbers from 1 to 19. Then, the variable “j” takes the value of the item inside the sequence individually and prints it repeatedly.

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Example 3:

If we want to create a sequence of numbers from 1 to 20 that means iteration starts from 1, and ends at 20 but increments by 2 instead of 1, so we write and print that:

```
for k in range(1,20,2):
    print(k)
```

Output:

```
1 3 5 7 9 11 13 15 17 19
```

So, we observe that iteration is incremented by two steps. As a result, 2, 4, 6… value was not shown in this sequence.

1.2. “for” loop with a list

A list is a type of container in Python's data structures, which is used to store multiple data at the same time. At first, we need to create a list of data by assigning a variable (list_item = ['jalal','Ripa', 2, 4, 6]).

If we need to execute the individual items of a list, we can loop through the list items by referring to their index number with the help of ‘range()’ and ‘len()’ functions to create a suitable iterable. The implementation of idea is given below:

```
list_item  = ['jalal','Ripa', 2, 4, 6]
 for i in range(len(list_item)):
    print(list_item[i])
```

From above syntaxes, we regard that the total number of items in “list_item” variable has been determined by “len” function which returns the length of a string or items. Then, the values of length are inputted into “range” function. As a result, the range function creates index values of list items. After that, “for” loop with variable “i” execute individual items of the list gradually.

Output:

```
jalal
Ripa
2
4
6
```

So, items on the list are printed individually. However, we can get the same result from the list of data without using “len” and “range” functions. Consequently, we can write the syntax again:

```
list_item  = ['jalal','Ripa', 2, 4, 6]
```

```
for i in list_item:
    print(i)
```

From syntaxes, we observed that every item of “list_item” variable (list of data) has been passed through “for” loop as the variable “i” item. So, we need to run these commands by “print” function, and the final result will be as same as before.

1.3. Looping through a string

In Python, the string is an immutable sequence data type. It is the sequence of ‘Unicode’ characters wrapped inside single, double, or triple quotes such as “Jalal”. We want to print all characters of a string. For reducing our effort and time, for loop will be used for iterating over a sequence of strings. So, we can write the syntaxes of this operation:

```
for i in "Jalal":

    print(i)
```

Here, “Jalal” is considered a string. The variable “i” receives the character directly from the string. Since this loop does not need the index value of each character, this loop format is very simple and preferred.

Output:

```
J
a
l
a
l
```

From this result, we can understand that all characters of string are printed individually through “for” loop.

1.4. “for” loop with a dictionary

In Python, dictionaries are one of the most useful and important data structures. Dictionaries are used to store data values in {key: value} pairs. It is created with curly brackets and has keys and values. While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique. We may explore a dictionary by using a ‘for’ loop. When looping through

a dictionary, the return value is the dictionary's keys, and other functions return the values as well. But now we will learn how to print all keys and values of the dictionary through "for" loop. First, we need to create a dictionary. So, the dictionary is:

```
mydict = {
  "name": "Jalal",
  "wife": "Ripa Jalal",
  "kids":"Jara Jalal",
  "year of birth": 1990,
  "good man":True,
  "favourite colors": ["white", "green", "red"]
 }
print(mydict)
```

Output:

```
{'name': 'Jalal', 'wife': 'Ripa Jalal', 'kids': 'Jara Jalal', 'year of birth': 1990, 'good man': True, 'favourite colors': ['white', 'green', 'red']}
```

We can access key and value pairs of all the items in the dictionary. To conduct this operation, "for" loop and "items()" functions of the dictionary will be used here. However, the "items()" method that returns a representation of the dictionary's contents as tuples. To iterate over the keys and values of "mydict" dictionary, we just need to 'unpack' the two items embedded in the tuple as shown below:

[Note: all values are converted to strings by "str()" function.]

```
for key, value in mydict.items():
    print(key + " : " + str(value))
```

Output:

```
name : Jalal
wife : Ripa Jalal
kids : Jara Jalal
year of birth : 1990
good man : True
favourite colors : ['white', 'green', 'red']
```

So, we have understood how we can easily print all keys and values of a dictionary by using “for” loop with ‘items()’ function.

1.5. Nested loop

In Python, a nested loop is a loop that is inside another loop. The “inner loop” will be executed one time for each iteration of the “outer loop”. In the nested loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop. Nested loops are commonly used to work with multidimensional data structures, such as printing two-dimensional arrays or iterating a nested list. The inner or outer loop can be any type, such as a ‘while’ loop or ‘for’ loop. For example, the outer ‘for’ loop can contain a ‘while’ loop and vice versa.

[Note: The outer loop can contain any number of the inner loop. There is no limitation on the nesting of loops.]

```
list_item1  = ['Jalal','Ripa', 'Jara']
list_item2  = ['Happy','Lucky', 'Honest']

 for i in list_item1:
    for j in list_item2:
       print(i,j)
```

From above syntaxes, we observe that we have two lists of data. “list_item1” consider an outer loop, and “list_item2” consider an inner loop.

- The outer ‘for’ loop have a list of item (list_item1) to iterate over the four items.
- The inner ‘for’ loop will execute four times for each outer number
- In the body of the inner loop, we will print the multiplication of the outer string and inner string.
- The inner loop is nothing but a body of an outer loop.

Output:

```
Jalal Happy
Jalal Lucky
Jalal Honest
Ripa Happy
Ripa Lucky
Ripa Honest
Jara Happy
Jara Lucky
Jara Honest
```

Another syntax of using a nested ‘for’ loop in Python:

```
a = [1,2,3,4,5]
b = [1,2,3,4,5]

 for i in a:
    for j in b:
        c = i**2 + j**2
        print(c)
```

From above syntaxes, we perceive that we have one inner loop but assign a new variable “c” with a new formula. So,

- The outer ‘for’ loop uses variable “a” (list of integers) to iterate over the one to five numbers.
- The inner ‘for’ loop will execute five times for each outer number
- In the body of the inner loop, we will print the multiplication of the square of the outer number and the square of an inner number

Output:

```
2
5
10
17
26
5
8
13
```

In python, the list “append()” method is used for appending and adding elements to the end of the list. Consequently, the syntaxes for adding of result to the list are:

```
a = [1,2,3,4,5]
b = [1,2,3,4,5]
result = []
for i in a:
    for j in b:
        c = i**2 + j**2
        result.append(c)

print(result)
```

Here, we have assigned a new variable of the list named “result”. Consequently, the final output of loop iteration will be added to the list variable (result) by “append()” function.

Output:

```
[2, 5, 10, 17, 26, 5, 8, 13, 20, 29, 10, 13, 18, 25, 34, 17, 20, 25, 32, 41, 26, 29, 34, 41, 50]
```

1.6. While loop with else

The ‘while’ loop is used in Python to repeatedly execute a block of statements until a given condition is fulfilled. Moreover, the line immediately after the loop is executed when the condition in the program becomes false.

Example:

```
i = 1
while i < 10:       # condition
    print(i)
    i += 1              # increment by 1

else:
    print("i is no longer less than 10")
```

From above syntaxes, we observe that this looping operation will be continued until 9 and increments of number are 1 (i +=1). But when the increased number is greater than or equal to 10, the condition is not satisfied and executed to the else condition.

Output:

```
1
2
3
4
5
6
7
8
9
i is no longer less than 10
```

2. Conditional Statements

The hard situations come in real life when we need to make some decisions and based on these decisions we decide what we should do next. Similar scenarios exist in programming when we must make decisions and then execute the following block of code depending on those decisions. Decision-making statements in programming languages decide the direction of the flow of program execution.

2.1. ‘if’ statement

The ‘if’ statement is used to decide whether a certain statement or block of statements will be executed or not.

2.2. ‘if-else’

The ‘else’ statement with ‘if’ statement is used to execute a block of code when the condition is false.

2.3. ‘if-elif-else’

The following example shows the application of the ‘if-elif-else’ statement.

```
a = 10


if a > 0:
   print("a is the positive number")
elif a <0:
```

```
    print("a is the negative number")
else:
    print("a is equal zero")
```

From these syntaxes, we realize that we have a variable with assigned values 10 (a=10). When we run it, we will see that "a is the positive number" because the value of the variable satisfied the first condition and execute its output. Otherwise, it will check below mentioned conditions and execute its result, respectively.

Output:

```
a is the positive number
```

2.4. Nested 'if' statement

The 'if', 'elif', and 'else' statements can be inserted within another 'if', 'elif', and 'else' statement.. The implementation of this idea is given below:

```
a = 10

if a > 0:
    print("a is the positive number")
    if a>5:                          # nested if
        print("a is the positive number")
```

Hence, we have an assigned value of 10 with variable "a". At first, if the value of the variable is satisfied with the first condition, then it will execute otherwise not. After satisfying the first condition, it will check the second condition. If the second condition is satisfied, it will execute its output otherwise not. Here, we observe that the value of the variable satisfies the first condition and executes its output, and then it also satisfies the nested condition and executes its output.

Output:

```
a is the positive number
a is the positive number
```

3. Break and continue statements in Python

Loops are used to automate and repeat processes more effectively. However, there may be occasions when you wish to exit the loop entirely, skip an iteration, or ignore the condition altogether. Loop control statements are useful for this. Loop control statements change execution from their normal sequence. Python provides break and continues statements to handle such situations and to have good control of your loop.

3.1. Break statements

The break statement is used to terminate the loop or statement in which it appears. After that, the control will pass to the statements that are present after the break statement, if are available.

Example:

Consider a situation where you want to iterate over a list of strings and want to print all the items until a string “Ripa” is encountered. It is specified that you have to do this using a loop and only one loop is allowed to use.

Here, comes the usage of the break statement. If it is “Ripa”, we will use the break statement to exit the loop.

```
list_item1  = ['Jalal','Ripa', 'Jara']

for i in list_item1:
   print(i)

   if i == 'Ripa':
      break
```

Output:

```
Jalal
Ripa
```

If we want to stop the iteration before “Ripa”, we need to change the placement of the print function. Under the loop, we need to place conditions with break statements, then have to place print function.

```
list_item1  = ['Jalal','Ripa', 'Jara']

for i in list_item1:
   if i == 'Ripa':
      break
   print(i)
```

Output:

```
Jalal
```

3.2. Continue statements

Like the break statement, continue is a loop control statement. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will activate.

Example 1:

Considering the situation, when you need to write a program which prints some strings such as “Jalal”, “Jara” and but not “Ripa”. It is specified that you have to do this using a loop and only one loop is allowed to use.

Here, comes the usage of the continue statement. What can we do here? we can run a loop from “Jalal”, “Ripa”, and “Jara” and every time we have to compare the value of the iterator with “Ripa”. If it is equal to “Ripa”, we will use the continue statement to continue the next iteration without printing anything; otherwise, we will print the value.

```
list_item1  = ['Jalal','Ripa', 'Jara']

for i in list_item1:
   if i == 'Ripa':
      continue
   print(i)
```

Output:

```
Jalal
```

Example 2:

Considering the situation in which we need to develop a program that prints numbers from 1 to 10 but not 6. It is mentioned that you must do this with a loop and that you may only use one loop at a time.

We can execute a loop from 1 to 10 and compare the value of the iterator with 6 at each step. If it equals 6, we will use the continue statement to skip to the next iteration without publishing anything else; otherwise, we will print the value.

The implementation of the aforesaid concept is shown below:

```
i = 1

while i < 10:
    print(i)

    if i == 6:
        break
    i += 1        # increments by 1
```

Output:

```
1
2
3
4
5
6
```

Chapter 7: Functions in Python

1. Functions in Python

Python functions are a collection of related statements that execute a computational, logical, or evaluative activity. The idea is to aggregate together some commonly or repeatedly done tasks and create a function, rather than writing the same code over and over for different inputs — we can call the function and reuse the code contained within it. Functions can be both built-in and user-defined. It helps the program to be concise, non-repetitive, and organized. For more information about the built-in function can be found online at https://docs.python.org/3/library/functions.html.

1.1. Defined Function

We create a Python function using the 'def' keyword. A colon (:) is being used to mark the end of the function header. After creating a function, we can call it by using the name of the function (function_name) followed by a parenthesis containing the parameters of that particular function.

```
def function_name(parameters):
    """function description"""
    statement(s)
 return expression
```

1.2. Arguments of Function

The values given within the function's parenthesis are referred to as arguments. A function can have any number of arguments separated by a comma.

1.3. Function description

The functions description string is the first string after the function. This is used to describe a function's functionality. The usage of function descriptions is not required; however, it is considered a good practice.

The below syntax can be used to print out the description of a function:

```
print(function_name.__doc__)
```

1.4. The return expression

The function return expression is used to leave a function and return the provided value or data item to the function caller.

```
return [expression_list]
```

The return expression can comprise an expression, a variable, or a constant which is returned at the end of the function execution. With the return expression, if none of the above is present, a 'None' object is returned.

1.5. How to call a function in Python?

We can call a function from another function, a program, or even the Python prompt once it has been defined. To use a function, simply type the name of the function followed by the relevant parameters.

Now, we will define a function that can calculate the average of values. Below is the implementation of the idea:

```
def avg (num):
    """
    Average = sum data / n

    """
    sum_data = 0
    for i in num:
        sum_data = sum_data + i
        avg_data = sum_data / len(num)
    return avg_data
```

The name of this above-mentioned defined function is "avg". So, we can call the function "avg". The descriptions of the function mentioned how we can calculate the average value of data. In the section of statements, step-by-step calculation procedures is added. Here, we have assigned a variable "sum_data", where the summation of all data will be assigned. Then, the loop plays an important role in the iteration of individual values from inputted values and is added with the previous value. In addition, variable "i" represent individual

inputted value. After summation, all values will be divided by counting the total inputted numbers with the help of "len()" function. After completion of the division, the final result will be displayed. Return is the last part of the defined function which will leave the program by giving the output. The implementation of this defined function is given below:

```
data = [1, 2, 3, 4, 5, 6]


avg_cal = avg(data)
print(avg_cal)
```

Here, a variable assigned named "data" with a list of numbers and the average of these numbers would be calculated by using the predefined function "avg()".

Output:

```
3.5
```

Now, we crosscheck this averaged value with the built-in function of Python. To determine the average value, first, we need to import the library of the "mean" function from the "statistics" module. The calculation procedures are given below:

```
from statistics import mean


avg_cal = mean(data)
print(avg_cal)
```

Here, the built-in "mean" function is used to calculate the average value of provided data. The output result of the built-in function and defined function are equal.

Output:

```
3.5
```

2. Wind speed calculation function

In meteorology, wind speed is a fundamental atmospheric quantity caused by air moving from high to low pressure, usually due to temperature changes. Wind speed is now commonly measured with an anemometer. Therefore, we will define a function to measure

wind speed. The name of the function is “windspeed” and the formula $\sqrt{(u^2 + v^2)}$will be used here. Below is the implementation of idea:

```
def windspeed(u,v):

    """Calculate wind speed

    Parameters
    ----------
    u : int or float
    v : int or float

    Returns
    ------
    speed : int or float
    """

    z     = (u*u) + (v*v)

    import numpy as np

    speed = np.sqrt(z)
    return speed
```

Here, the name of the defined function is “windspeed”. From the function description, we observe the types of parameters (u and v) are being used here. Variable “z” is assigned with the formula of (u^2+v^2). After that, speed would be executed by square root over the “z” variable. Then, the function “return” will leave the function and return the provided value. If we install “MetPy” library, we do not need to develop a function. We can install this library by running the code: “**pip install metpy --user**”. More information can be found online at

https://unidata.github.io/MetPy/latest/userguide/installguide.html and https://unidata.github.io/MetPy/latest/api/generated/metpy.calc.wind_speed.html.

Now, we can use the built-in function “wind_speed” for determining wind speed. For your kind information, the source codes of the “wind_speed” function are given below:

```
import numpy as np

def wind_speed(u, v):
    r"""Compute the wind speed from u and v-components.

    Parameters
    ----------
    u : `pint.Quantity`
        Wind component in the X (East-West) direction
    v : `pint.Quantity`
        Wind component in the Y (North-South) direction

    Returns
    -------
    wind speed: `pint.Quantity`
        The speed of the wind

    See Also
    --------
    wind_components

    """
    speed = np.sqrt(u * u + v * v)
    return speed
```

Below is the implementation of the defined and built-in function of wind speed calculation.

At first, we will use a defined function named “windspeed” for the determination of wind speed. After that, we will utilize the built-in function.

```
U = 10
V = 20

winspd = windspeed(U,V)
print(winspd)
```

Here, U and V variables are assigned with a scalar value of wind speed. After inputting the values of variables into a defined function by calling “windspeed()”, the final output is mentioned below.

Output:

```
22.360679774997898
```

The same result will also be executed by the implementation of the built-in function “wind_speed()”. So, the codes are given below:

```
winspd = wind_speed(U,V)
print(winspd)
```

Output:

```
22.360679774997898
```

So, we observe that the final results of wind speed are equal in defined and built-in functions. However, what can we do if the provided data is a vector? Previous calculation of wind speed was conducted on scalar data. We have mentioned below how to measure the wind speed of vector data by calling a defined function.

```
data1 = [1, 2, 3, 4, 5, 6]
data2 = [1, 2, 3, 4, 5, 6]

wspeed = []

for i in data1:
```

```
    for j in data2:
        winspd = windspeed(i,j)
        wspeed.append(winspd)



print("Wind speed: \n", wspeed)
```

Here, we have two lists of vector data with assigned variables such as data1 and data2. To save the output result, we needed to take a new variable named “wspeed” which is a list type but empty. For interpreting all data in the calculation, we have assigned “data1” in the outer “for” loop and “data2” in the inner “for” loop. Variable “i” will take individual value from data1 and variable “j” will take individual value from data2. Then, the values of “i” and “j” will be put into a defined function named “winspeed()”. The output of this calculation will be added to a variable named “wspeed” which is the list type.

Output:

```
Wind speed:
 [1.4142135623730951, 2.23606797749979, 3.1622776601683795, 4.123105625617661, 5.0990195135927845, 6.082762530298219, 2.2360679
7749979, 2.8284271247461903, 3.605551275463989, 4.47213595499958, 5.385164807134504, 6.324555320336759, 3.1622776601683795, 3.6
05551275463989, 4.242640687119285, 5.0, 5.830951894845301, 6.708203932499369, 4.123105625617661, 4.47213595499958, 5.0, 5.65685
4249492381, 6.4031242374328485, 7.211102550927978, 5.0990195135927845, 5.385164807134504, 5.830951894845301, 6.403124237432848
5, 7.0710678118654755, 7.810249675906654, 6.082762530298219, 6.324555320336759, 6.708203932499369, 7.211102550927978, 7.8102496
75906654, 8.48528137423857]
```

1. Data analysis with Pandas

After opening the Jupyter Notebook, we need to follow the given steps:

- The following command is for importing 'numpy' and 'pandas' libraries:

```
import numpy as np
import pandas as pd
```

[N.B. 'Numpy' should be imported for array management and 'pandas' for excel, .csv files, and similar types of data handling and management.]

1.1. Creating a series in Pandas

A series is a one-dimensional array of any data i.e., consisting of only one column. Before creating a series, we can create data by 'arange()' function of 'numpy' library by the following command:

```
x = np.arange(0,6,1)

x
```

[N.B. The command can be expressed as follows –

x = np.arange(first value, last value, step/increment)]

Output:

```
array([0, 1, 2, 3, 4, 5])
```

We get six values from 0 to 5 as Python is zero-based programming language. After that, we can create a list of 'x' by the following commands using 'list()' function/ using square brackets:

```
y = list(x)

y
```

Output:

```
[0, 1, 2, 3, 4, 5]
```

A series can be formed by ‘Series()’ function of the pandas library. The series for ‘y’ list can be created by the given command:

```
ser = pd.Series(y)

print(ser)
```

Output:

```
0    0
1    1
2    2
3    3
4    4
5    5
dtype: int64
```

[N.B. The first column is for the line/ row numbers and the second column is for the created series.]

Now, we can give a name instead of the row/line numbers by the ‘index’ argument in ‘Series()’ function:

```
ser1 = pd.Series(y, index = ["Jalal", "Babul", "Tamim", "Emdad", "Hazera", "Nazia"])

print(ser1)
```

Output:

```
Jalal     0
Babul     1
Tamim     2
Emdad     3
Hazera    4
Nazia     5
dtype: int64
```

[N.B. In python, ‘index’ means row and ‘column’ indicates column.]

The series name and index name can also be assigned by following commands:

```
ser1.name       = "Rank of the best student"

ser1.index.name = "Name of the students"

print(ser1)
```

Output:

```
Name of the students
Jalal     0
Babul     1
Tamim     2
Emdad     3
Hazera    4
Nazia     5
Name: Rank of the best student, dtype: int64
```

1.2. Creating a dataframe in Pandas

Two-dimensional data structures are called data frames which consist of tables with rows, columns, and data. The dictionary contains keys and values. We can make a nested dictionary consisting of three dictionaries by the following commands:

```
training = {

 "Research Society" : {

  "course name" : "Python",

  "participants level":"all",

  "year" : 2020

 },



 "IMSA" : {

  "course name" : "Matlab",

  "participants level":"all",

  "year" : 2020

 },



 "CUSS and RS" : {
```

```
   "course name" : "Statistics",
   "participants level":"all",
   "year" : 2020
  }
}
print(training)
```

[N.B. Three dictionaries named 'Research Society', 'IMSA', and 'CUSS and RS' contain several courses' names, the level of participants eligible for the courses, and the year of occurrence.]

Output:

```
{'Research Society': {'course name': 'Python', 'participants level': 'all', 'year': 2020}, 'IMSA': {'course name': 'Matlab', 'participants level': 'all', 'year': 2020}, 'CUSS and RS': {'course name': 'Statistics', 'participants level': 'all', 'year': 2020}}
```

We can make a data frame of the 'training' dictionary (nested) by running the following commands using the 'DataFrame()' function of the pandas library:

```
df = pd.DataFrame(training)
df
```

Output:

	Research Society	IMSA	CUSS and RS
course name	Python	Matlab	Statistics
participants level	all	all	all
year	2020	2020	2020

We can also create a single dictionary by the following command:

```
Research_Society = {
   "course name" : ["Python","MATLAB","NCL"],
```

```
   "participants level":["Undergrade","Masters","PhD"],

  }

print(Research_Society)
```

[N.B. The dictionary named 'Research_Society' contains several courses' names provided, and the level of participants eligible for the respective courses.]

Output:

```
{'course name': ['Python', 'MATLAB', 'NCL'], 'participants level': ['Undergrade', 'Masters', 'PhD']}
```

We can make a data frame for the 'Research_Society' dictionary by running the following commands using the 'DataFrame()' function of the pandas library:

```
df1 = pd.DataFrame(Research_Society)

df1
```

Output:

	course name	participants level
0	Python	Undergrade
1	MATLAB	Masters
2	NCL	PhD

Now, we can give the name of the row/line numbers of the data frame by the 'index' argument:

```
df1 = pd.DataFrame(Research_Society,index = [2019,2020,2021])

df1
```

Output:

	course name	participants level
2019	Python	Undergrade
2020	MATLAB	Masters
2021	NCL	PhD

We can make a tuple consisting of two lists with numeric and string variables by the following commands:

```
data2 = ([1,2,3,4],["A","B","C","D"])

data2
```

Output:

```
([1, 2, 3, 4], ['A', 'B', 'C', 'D'])
```

Now, we can make a data frame for this tuple as follows:

```
data2 = pd.DataFrame(data2)

data2
```

Output:

	0	1	2	3
0	1	2	3	4
1	A	B	C	D

We can also add names for the rows and columns using the 'index' and 'columns' argument in DataFrame() function for a new tuple named 'data3':

```
data3 = ([1,2,3,4],["A","B","C","D"])

data3

data3 = pd.DataFrame(data3, columns = ["Jalal", "Babul", "Tamim", "Emdad"], index = ["Rank","Grade"])

data
```

The output will contain the column's name as students' names, and rows will be indexed as rank and grade of the students, respectively.

Output:

	Jalal	Babul	Tamim	Emdad
Rank	1	2	3	4
Grade	A	B	C	D

This data frame can be saved as excel or .csv files by using the ‘to_excel()’, and ‘to_csv()’ commands:

```
data3.to_excel("class_pandas.xlsx")

data3.to_csv("class_pandas.csv")
```

[N.B. The commands can be expressed as follows –

```
dataframe_name.to_excel(“file_name”,extension)
dataframe_name.to_csv(“file_name”,extension]
```

1.3. Importing excel or .csv files

We can open an excel or .csv file using the ‘read_excel()’ or ‘read_csv()’ function of the pandas library by using the following commands:

```
import pandas as pd

import numpy as np

data = pd.read_excel("data.xlsx")

data
```

We will see a dataset in the output consisting of a year, pet (potential evapotranspiration), and di for climatic water balance monitoring. By default, Python prints only the first sheet of excel/ .csv files after using the ‘read_excel()’ or ‘read_csv()’ functions from the pandas library.

	year	pet	di
0	1980	1563.052578	12.355643
1	1981	1421.942566	818.463441
2	1982	1504.641507	-402.249164
3	1983	1487.846696	140.541067
4	1984	1493.770060	80.621273
5	1985	1485.745039	-234.332012
6	1986	1473.358766	36.058598
7	1987	1576.478339	-90.054706
8	1988	1533.748913	49.678843
9	1989	1527.662614	-203.282885
10	1990	1440.674072	325.728321
11	1991	1499.803192	-2.407143
12	1992	1537.114096	-694.735046
13	1993	1469.711139	152.701099
14	1994	1503.897981	-362.499563
15	1995	1581.972082	-150.585899
16	1996	1554.242977	-285.841300
17	1997	1438.116497	623.270135

We need to use the ‘sheet_name’ argument for importing a particular sheet other than the first one from excel or .csv file. In that case, we can imitate the following commands to print a specific sheet of excel or .csv file:

```
data1 = pd.read_excel("data.xlsx",sheet_name = "pet")

data1
```

Output:

	year	pet
0	1980	1563.052578
1	1981	1421.942566
2	1982	1504.641507
3	1983	1487.846696
4	1984	1493.770060
5	1985	1485.745039

We can observe the information of the only first five rows of the dataset by using the ‘head()’ function:

```
data1.head()
```

Output:

	year	pet
0	1980	1563.052578
1	1981	1421.942566
2	1982	1504.641507
3	1983	1487.846696
4	1984	1493.770060

The size (row and column) numbers of the dataset can also be obtained by using the 'shape' function:

```
data1.shape
```

The output will show the shape of the dataset with row and column numbers.

Output:

```
(37, 2)
```

Descriptive statistics, including mean, standard deviation, minimum value, maximum value, and percentiles can be attained by using the 'describe()' function:

```
data1.describe()
```

The output will contain the basic statistics of the dataset

Output:

	year	pet
count	37.000000	37.000000
mean	1998.000000	1536.174969
std	10.824355	60.735851
min	1980.000000	1421.942566
25%	1989.000000	1493.770060
50%	1998.000000	1533.748913
75%	2007.000000	1575.113087
max	2016.000000	1705.572793

1.4. Use of .loc and .iloc functions

We can use '.loc[]' and '.iloc[]' functions to access data in any dataset. .loc (location) function locates data by the index name (i.e., selecting by label) and .iloc(index location) function locates any data by numerical index (i.e., selecting by position). We can print only the first six years' 'pet' values by using the '.loc' function or by providing the labels of the rows and columns:

```
data1.loc[0:5,['year','pet']]
```

The output will provide the first six 'year', and 'pet' values, where '0:5' means rows 1 to 6.

Output:

	year	pet
0	1980	1563.052578
1	1981	1421.942566
2	1982	1504.641507
3	1983	1487.846696
4	1984	1493.770060
5	1985	1485.745039

On the other hand, we can use the '.iloc' function to locate the data using numerical index values:

```
data1.iloc[1]
```

The output will print the value of the second row and second column, as python is 0 based language.

Output:

```
year    1981.000000
pet     1421.942566
Name: 1, dtype: float64
```

We can get the value of the second column of the first row by the following code:

```
data1.iloc[0,1]
```

[N.B. Here, the command can be expressed as: dataset.iloc[row or index number, column number]

The output will give only the value of the second column in the first row.

Output:

```
1563.052577562614
```

We can access a range of datasets by using the slicing operator [:] inside the '.iloc' function:

```
data1.iloc[0:6,0:6]
```

[N.B. Here, the command can be expressed as: dataset.iloc[row or index range, column range]]

The output will print the first six-row and column values from the dataset.

Output:

	year	pet
0	1980	1563.052578
1	1981	1421.942566
2	1982	1504.641507
3	1983	1487.846696
4	1984	1493.770060
5	1985	1485.745039

1.5. Data merging

We can merge two datasets by simply using the 'merge()' function of the pandas' library. However, if we have the same column in both datasets, the columns will be replaced and merged into one column in the new dataset:

```
new_data = pd.merge(data,data1)

new_data.head()
```

We can merge 'data' and 'data1' where data has two common columns 'year', and 'pet' as data1. But data has one new column named 'di' than data1. So, the new dataset will merge and consist of three columns named 'year', 'pet', and 'di'. We can observe the first five rows of the 'new_data' using the 'head()' function.

Output:

	year	pet	di
0	1980	1563.052578	12.355643
1	1981	1421.942566	818.463441
2	1982	1504.641507	-402.249164
3	1983	1487.846696	140.541067
4	1984	1493.770060	80.621273

1.6. Data concatenation

We can use ‘concat()’ function of the pandas' library to combine two datasets. However, we need to make a list of the datasets to be combined inside the ‘concat()’ function using the indexing operator (square bracket):

```
new_data1 = pd.concat([data,data1])

new_data1.head()
```

[N.B. Here, the command can be expressed as dataset_name = pd.concat([first dataset, second dataset])]

We can observe the first five rows of the ‘new_data1’ dataset using the ‘head()’ function.

Output:

	year	pet	di
0	1980	1563.052578	12.355643
1	1981	1421.942566	818.463441
2	1982	1504.641507	-402.249164
3	1983	1487.846696	140.541067
4	1984	1493.770060	80.621273

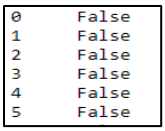
1.7. Data duplicating

Checking and removing duplicate values from the dataset is an important task in case of data cleaning. We can check any duplicate values in a dataset by using the ‘duplicated()’ function with an argument called ‘subset’. We have to input the column names among which we want to check duplicate values in the ‘subset’ argument:

```
data.duplicated(subset=['pet','di'])
```

The output will give us the results in Boolean values (False, or True).

Output:



We can remove the duplicate values found using the 'drop_duplicates()' function:

```
data.drop_duplicates(subset=['pet','di'])
```

The output will print the dataset after removing the duplicate values.

Output:

	year	pet	di
0	1980	1563.052578	12.355643
1	1981	1421.942566	818.463441
2	1982	1504.641507	-402.249164
3	1983	1487.846696	140.541067
4	1984	1493.770060	80.621273
5	1985	1485.745039	-234.332012

[N.B. There was no duplicate value in the 'data' dataset. Hence, the original dataset was printed in the output after using the 'drop_duplicates()' function.]

1. Data visualization in Python (temporal & time series plots)

After opening the Jupyter Notebook, we need to follow the given steps:

- The following command is for importing necessary libraries:

```
import pandas as pd

import matplotlib.pyplot as plt
```

[N.B. 'Pandas' library should be imported for excel, .csv files, and similar types of data handling and management. 'Pyplot' module of the 'Matplotlib' library is required for data visualization and plotting.]

We can open the dataset 'data.xlsx' file using the 'read_excel()' function of the pandas' library by using the following commands:

```
data = pd.read_excel("data.xlsx", sheet_name = "pet")

data.head()
```

We will see a dataset in the output consisting of a year, pet (potential evapotranspiration) values for climatic water balance monitoring. By default, python prints only the first sheet of excel/ .csv files after using the 'read_excel()' or 'read_csv()' functions from the pandas' library. However, we need to use the 'sheet_name' argument for importing a particular sheet other than the first one from excel or .csv file. We can also observe the information of the only first five rows of the dataset by using the 'head()' function.

The output will print the first five rows of the sheet named 'pet' from the 'data.xlsx' excel file.

Output:

	year	pet
0	1980	1563.052578
1	1981	1421.942566
2	1982	1504.641507
3	1983	1487.846696
4	1984	1493.770060

1.1. Line chart and time series plot

Any dataset relating to time (e.g., second, minute, half-hourly, hourly, weekly, monthly, and annually, etc.) can be called time-series data. Line charts can be used to visualize the time series data. We can create a line chart of the 'year' and 'pet' variables in the 'data.xlsx' dataset by using the 'plot()' function from the 'pyplot' module of the 'matplotlib' library:

```
plt.plot('year', 'pet', data=data, color = 'red')
```

[N.B. The command can be expressed as follows –

plt.plot('xlabel', 'ylabel', data = obj or object name, color = 'line color')]

Output:

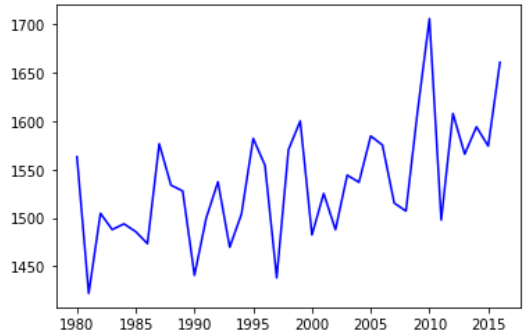


Figure 9.1. Line chart

We can also create a similar type of line chart with blue circle markers by using the following commands:

```
plt.plot(data['year'],data['pet'],'b--o')
```

[N.B. The command can be expressed as follows –

plt.plot(dataset['xlabel'],dataset['ylabel'], 'color line-style marker-shape')]

Output:

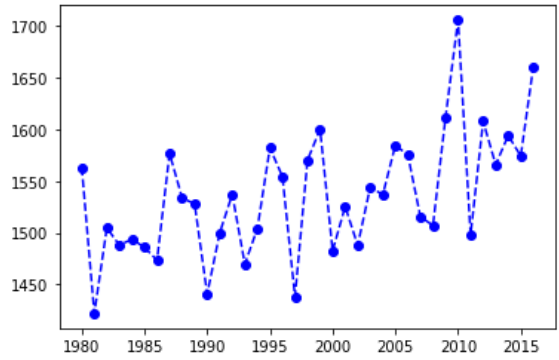


Figure 9.2. Line chart with markers

We can add line width and marker size with the previous command:

```
plt.plot(data['year'],data['pet'],'b--o', linewidth=2, markersize=10)
```

Output:

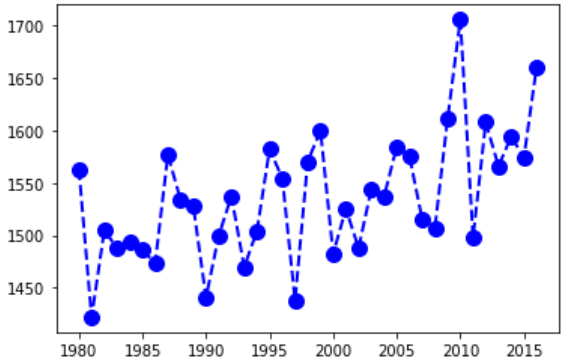


Figure 9.3. Line chart with line width and marker size modifications

The 'b--o' code can be written in detail as the following command to get a similar line chart with green line colour instead of blue:

```
plt.plot(data['year'],data['pet'], color='green', marker='o', linestyle='dashed', linewidth=2, markersize=10)
```

Output:

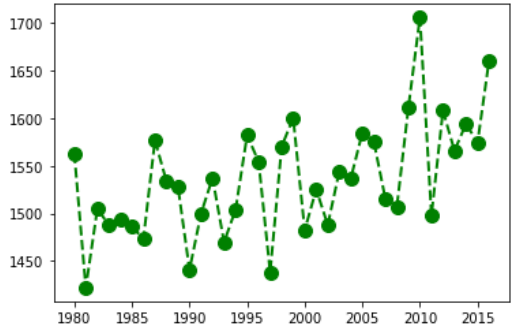


Figure 9.4. Line chart with modifications in colour

We will open the ‘pet_di’ sheet of the dataset ‘data.xlsx’ using the ‘sheet_name’ argument in the ‘read_excel()’ function of the ‘pandas’ library with the following commands:

```
data1 = pd.read_excel("data.xlsx", sheet_name = "pet_di")

data1.head()
```

If we want to plot both ‘pet’ and ‘di’ variables in the y-axis against the ‘year’ in the x-axis, we can run the following code:

```
plt.plot(data1['year'],data1['pet'], 'b--^', data1['year'],data1['di'], 'r-o')
```

[N.B. The command can be expressed as follows –

plt.plot(dataset[‘xlabel1’],dataset[‘ylabel1’], ‘color line-style marker-shape’, dataset[‘xlabel2’],dataset[‘ylabel2’], ‘color line-style marker-shape’)]

Output:

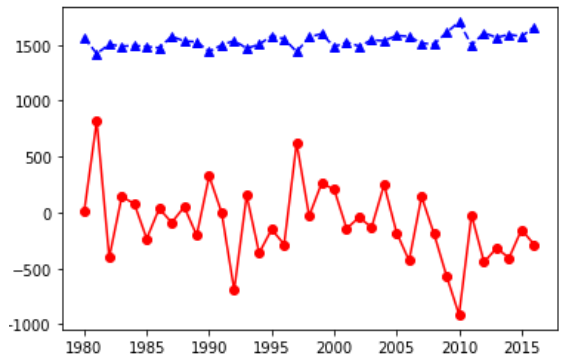


Figure 9.5. Line charts with two variables

We need to determine the width and height of the figures (in inches) before plotting and exporting. We have used a minimum figure dpi (80) for plotting. However, maximum journals require a standard figure resolution of 300 dpi. So, we can save the figures in high a resolution (300 dpi) by the following commands:

```
plt.figure(figsize=(6,4), dpi= 80)

plt.plot(data['year'],data['pet'], color='green', marker='o', linestyle='dashed', linewidth=2, markersize=10)

plt.xlabel("Year")

plt.ylabel("PET (mm)")

plt.title("Yearly PET")

plt.savefig('Yearly_pet.png',dpi=300)
```

Before saving the plot, we added the name of the x and y axes by using 'xlabel()' and 'ylabel()' functions. 'title()' function has been used for providing a title for the plot.

Output:

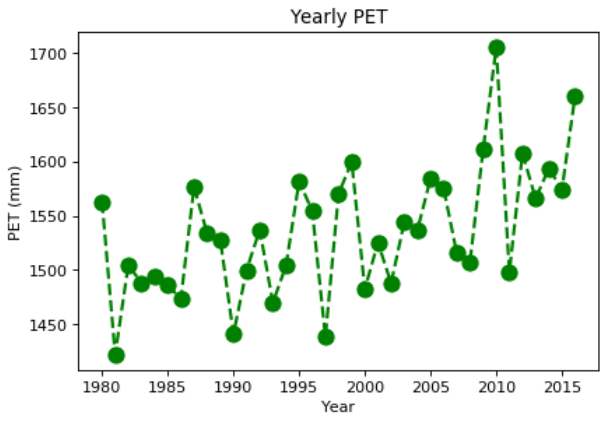


Figure 9.6. Line charts for time series plot

[N.B. The following websites can provide more information regarding plots and figures with 'matplotlib.pyplot' library in python:

- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.figure.html]

1.2. Scatter plot

Scatter plots are used to show how much one variable is affected by another. Scatter plots are mainly used in the visualization of correlations between variables. We will open the 'pet_di' sheet of the dataset 'data.xlsx' using the 'sheet_name' argument in the 'read_excel()' function of the 'pandas' library with the following commands:

```
pet_di = pd.read_excel("data.xlsx",sheet_name = "pet_di")

pet_di.head()
```

Output:

	year	pet	di
0	1980	1563.052578	12.355643
1	1981	1421.942566	818.463441
2	1982	1504.641507	-402.249164
3	1983	1487.846696	140.541067
4	1984	1493.770060	80.621273

We can get a scatter plot by placing both 'pet' and 'di' variables in the y-axis against the 'year' in the x-axis using the 'scatter()' function of the 'matplotlib.pyplot' library by running the following code:

```
plt.scatter(pet_di['year'], pet_di['pet'], color='green', label = 'PET')

plt.scatter(pet_di['year'], pet_di['di'], color='red', label = 'DI')

plt.xlabel("Year")

plt.ylabel("PET and DI (mm)")

plt.legend()
```

[N.B. The command for bar plotting with labels can be expressed as follows –

```
plt.bar(dataset('xvariable1'), dataset('yvariable1'), color = 'name', label = 'label'
```

```
        plt.bar(dataset('xvariable1'), dataset('yvariable2'), color = 'name', label = 'label'
]
```

Output:

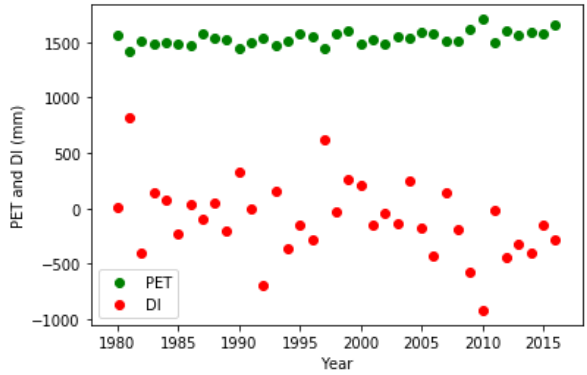


Figure 9.7. Scatter plot

1.3. Single bar plot

Bar plots can be used to compare various samples, groups or categories of variables in a dataset. Here, we can make two lists of two variables consisting of students' names and their exam marks. We can compare the marks of the students by plotting these variables in a single bar plot using the 'bar()' function of the 'matplotlib.pylot' library:

```
student_name  =  ['Jalal', 'Jara', 'Babul', 'Tamim', 'Emdad']

mark          =  [95,90,85,70,65]

plt.bar(student_name,mark)

plt.xlabel('Name of students')

plt.ylabel('Exam score')

plt.show()
```

Output:

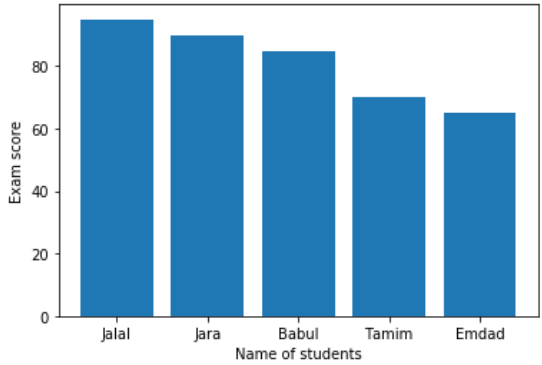


Figure 9.8. Single bar plot

1.4. Multiple bar charts

Multiple bar charts can be used to compare various groups with different categories. We can make three marks lists for three subjects over four years. In that case, we can make a multiple bar chart with four groups representing four years and each group contains three bars indicating three subjects. For creating four positions for four years, 'np.arange()' function can be used. Multiple bar charts are plotted based on thickness and the positions of the bars. In this case, the thickness of each bar will be given as 0.25 units using the 'width' argument. Each bar chart will be shifted 0.25 units from the previous one by using the following commands:

```
import numpy as np

English    = [60, 75, 80, 30]

Mathematics = [17, 20, 50, 45]

Physics    = [70, 50, 75, 30]

x = np.arange(4)

plt.bar(x + 0.00, English, color = 'b', width = 0.25, label = 'English')

plt.bar(x + 0.25, Mathematics, color = 'r', width = 0.25, label = 'Mathematics')

plt.bar(x + 0.50, Physics, color = 'g', width = 0.25, label = 'Physics')
```

```
plt.legend()

plt.xticks([0 + 0.25, 1 + 0.25, 2 + 0.25, 3 + 0.25],[2017, 2018, 2019, 2020])
```

[N.B. The commands for adding ticks in the middle position for each group or year in the x-axis can be expressed as follows:

plt.xticks([group position number 1+ 0.25, group position number 2 + 0.25, group position number 3 + 0.25, group position number 4 + 0.25],[label1, label2, label3, label4])

Here, 0.25 after the position number in the ticks defines the positions of the ticks in each group.]

Output:

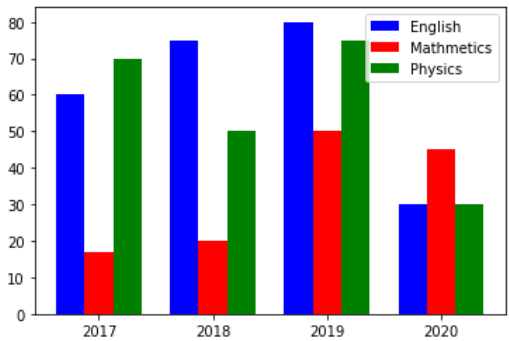


Figure 9.9. Multiple bar charts with legends

1.5. Stacked bar chart

In a stacked bar chart, the value of one group will go above the value of another group instead of running from zero for each bar. This will be suitable when we have no value for some variables or we need to visualize the proportion of percentages depicted by each category. Here, we can use the previous example of making stacked bar charts for four years with two categories/ subjects in each group. We have to run one by one code for each category. However, we need to use the 'bottom' argument in the 'bar()' function to specify the base category's bar chart as follows:

```
plt.bar(x, English, color = 'b', width = 0.25, label = 'English')

plt.bar(x, Mathematics, color = 'r', width = 0.25, bottom = English, label = 'Mathematics')
```

```
plt.legend()

plt.xticks([0, 1, 2, 3],[2017, 2018, 2019, 2020])
```

[N.B. The commands for adding ticks in the middle position for each group or year in the x-axis can be expressed as follows:

```
plt.xticks([group position number 1, group position number 2, group position number 3, group position number 4],[label1, label2, label3, label4])
```

Or simply,

```
plt.xticks([ticks],[labels])
```

]

Output:

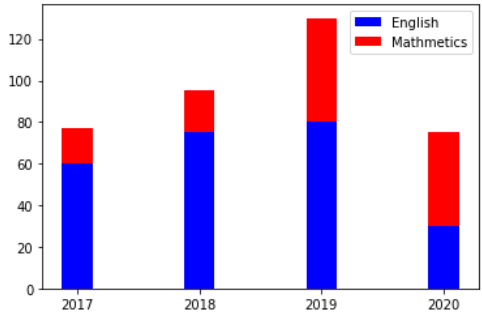


Figure 9.10. Stacked bar charts with legends

1.6. Histogram

The histogram is used mostly for statistical analysis purposes, which require bins, cumulative, probability density function, frequency distribution etc. We can make a list of scores for five participants and then create a histogram using the 'hist()' function and by providing some specific bins using the 'bins' argument with the following commands:

```
score    = [60, 75, 80, 30,17, 20, 50, 45,70, 50, 75, 30]

plt.hist(score , bins = [10,25,50,75,100])

plt.xlabel('Exam score')

plt.ylabel('Number of participants')

plt.yticks([0, 1, 2, 3, 4], [0, 1, 2, 3, 4])
```

[N.B. Here, the command for ticks' position in the y-axis can be expressed as:

```
plt.yticks([ticks' positions], [labels])
```

]

Output:

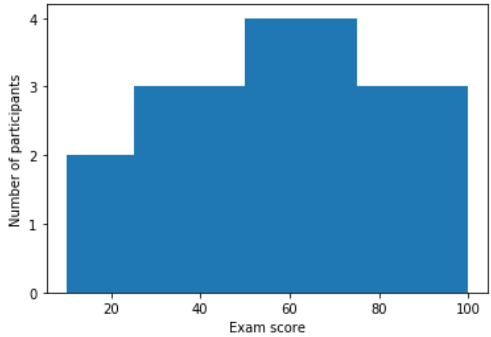


Figure 9.11. Histogram

1.7. Pie chart

The pie chart can be used to visualize the proportionate percentages of the data in respective segments. Here, we can make two lists of two variables consisting of students' names and their exam marks. We can make a pie chart to display each student's contribution among 100% marks using the 'pie()' function:

```
student_name   = ['Jalal', 'Jara', 'Babul', 'Tamim', 'Emdad']

mark           = [95,90,85,70,65]

plt.pie(mark, labels = student_name, autopct='%.1f%%')

plt.show()
```

[N.B. Here, the command for pie chart along with the labelling of each proportionate percentage inside each respective segment can be expressed as:

```
plt.pie(data, labels = label_name, autopct='fmt%pct')
```

Here, 'fmt' is equal to formatting (like % % as value), '.1' indicates one decimal value will be displayed and % indicates the symbol of percentage as a string.]

Output:

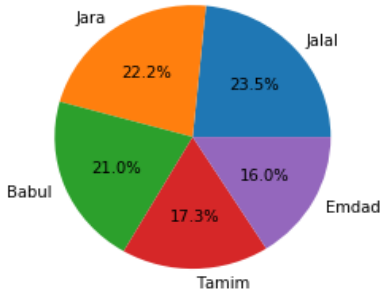


Figure 9.12. Pie chart

1.8. Subplot

Subplots are also known as panel plots where we can make a figure consisting of more than one plot. Subplots can be created using the 'subplot()' function with four arguments as 'subplot(nrows, ncols, index, **kwargs)', where nrows, ncols and index are the number of rows, number of columns, and position of the plot, respectively. By default, the arguments are taken as (1, 1, 1) in python. The index starts at 1 in the upper left corner and increases to the right. We can make two subplots of 'year' vs 'pet' and 'year' vs 'di' from the 'data.xlsx' dataset by the following commands:

```
plt.subplot(211)

plt.plot(data['year'],data['pet'], color='green', marker='o', linestyle='dashed', linewidth=2,
        markersize=10)

plt.subplot(212)

plt.plot(data1['year'],data1['di'], color='red', marker='o', linestyle='dashed', linewidth=2,
        markersize=10)
```

[N.B. Here, the command for subplots can be expressed as:

```
plt.subplot(211)
```

Here, 211 = two no. of rows, one no. of the column, and the position of the first plot is 1 (upper left).]

Output:

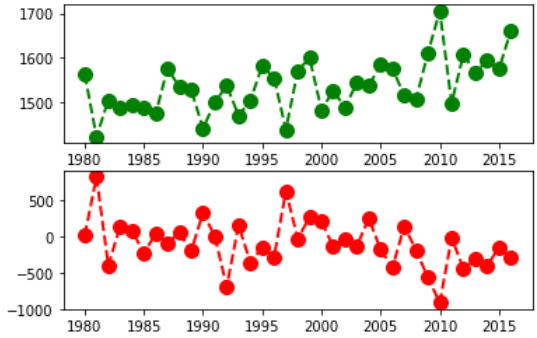


Figure 9.13. Subplots

These same subplots can be created by using some alternative commands as follows:

```
fig, (ax1, ax2) = plt.subplots(2, 1)

fig.subplots_adjust(hspace=0.5)

ax1.plot(data['year'],data['pet'], color='green', marker='o', linestyle='dashed',

            linewidth=2, markersize=10)

ax1.set_xlabel('Year')

ax1.set_ylabel('PET (mm)')

ax2.plot(data1['year'],data1['di'], color='red', marker='o', linestyle='dashed',

             linewidth=2, markersize=10)

ax2.set_xlabel('Year')

ax2.set_ylabel('DI (mm)')

plt.savefig("PET_DI.png",dpi=300)

plt.show()
```

[N.B. Here, the command for subplots can be expressed as:

```
fig, (ax1, ax2) = plt.subplots(2, 1)
```

(2, 1) = Two no. of rows, one no. of column

```
fig.subplots_adjust(hspace=0.5)
```

This command is to make a little extra horizontal space between the subplots.]

Output:

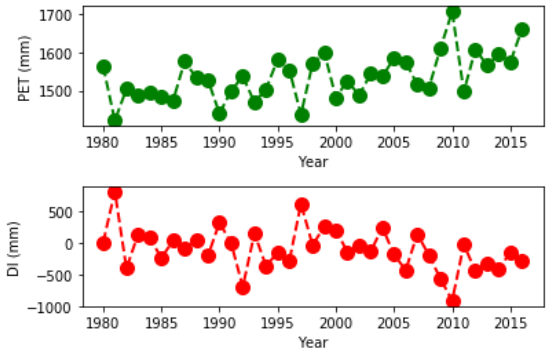


Figure 9.14. Subplots with extra horizontal space in between

We need to visualize sometimes two variables on the y-axis against the x-axis variable in the same plot. However, the example in the line charts and scatter plot was not comprehensible enough. Variables with different ranges or different units can be displayed against one variable (x-axis) in the same plot using both the left and right y-axis of the plot. In that case, we can use 'twinx()' function to make and return a second axis that share the x-axis:

```
fig, ax1 = plt.subplots(1, 1, figsize=(10,4), dpi= 80)

ax1.plot(data['year'],data['pet'], color='green', marker='o', linestyle='dashed',

            linewidth=2, markersize=10, label = 'PET')

ax1.set_xlabel('Year')

ax1.set_ylabel('PET (mm)')

ax1.legend(loc='upper center')

ax2 = ax1.twinx()

ax2.plot(data1['year'],data1['di'], color='red', marker='o', linestyle='dashed',

            linewidth=2, markersize=10, label = 'DI')
```

```
ax2.set_xlabel('Year')

ax2.set_ylabel('DI (mm)')

ax2.legend(loc='upper right')

fig.tight_layout()

plt.savefig("secondary_Y_axis.png",dpi=300)

plt.show()
```

[N.B. Here, the command for secondary axis subplots can be expressed as:

```
ax2 = ax1.twinx()
```

The command is for plot line 2 on the right y-axis. This means a secondary y-axis will be created alongside the primary y-axis against the same x-axis.

The following websites can provide more information regarding subplots with 'matplotlib.pyplot' library in python:

- https://matplotlib.org/3.5.0/api/_as_gen/matplotlib.pyplot.subplot.html]

The output will give a subplot with the right and left y-axes of two variables against the x-axis variable (primary and secondary axes).

Output:

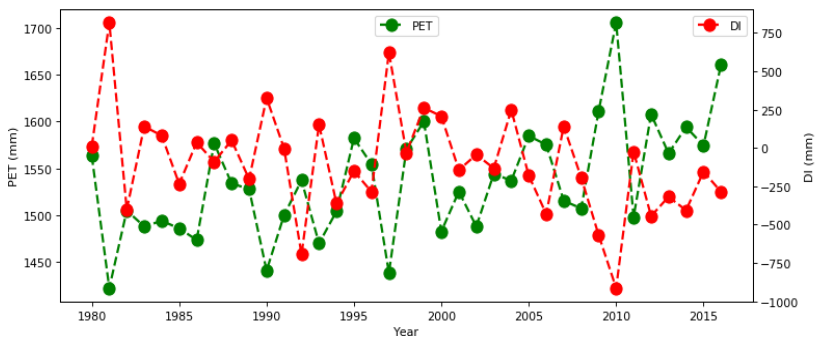


Figure 9.15. The subplot of two variables against the x-axis

1.9. Contour plots

Contour plots can be used when we have three-dimensional data (or more than two-dimensional data) e.g., Z = f(X, Y), where Z value changes as a function of two inputs, X

and Y. We can open a dataset named ‘MSW.xlsx’ which contains three-dimensional data with latitude (°N), longitude (°E), and maximum sustained wind speed (m s^{-1}). We can import the necessary libraries and the dataset by the following commands:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

msw = pd.read_excel("MSW.xlsx")

msw.head()
```

We can observe the first five rows of the dataset in the output as follows.

Output:

	Year	Month	Day	Hour	Latitude (°N)	Longitude (°E)	MSW (ms-1)
0	2015	1	14	0	10.0	139.8	18
1	2015	1	14	6	10.4	138.4	18
2	2015	1	14	12	11.0	137.2	18
3	2015	1	14	18	11.3	135.8	18
4	2015	1	15	0	11.5	134.7	20

The shape of the dataset (rows and column numbers) can be observed by the following command:

```
msw.shape
```

Output:

```
(200, 7)
```

Before contour plotting, the dataset variables need to be converted into two dimensions using ‘meshgrid()’ function:

```
latitude  = msw['Latitude (°N)']

longitude = msw['Longitude (°E)']

MSW       = msw['MSW (ms-1)']
```

```
lat, lon       = np.meshgrid(latitude, longitude)

MSW_ms, lat_= np.meshgrid(MSW, latitude)

plt.contourf(lon, lat, MSW_ms)

plt.colorbar()

plt.xlabel('longitude')

plt.ylabel('latitude')

plt.xlim(140,152)

plt.ylim(8,36)
```

[N.B. Here, the command for contour plotting can be expressed as:

```
plt.contourf(X, Y, Z)
```

At first, 'latitude' and 'longitude' will be converted to two-dimension. Then, 'MSW' will also be converted into 2D using 'meshgrid()' function from 'numpy' library. 'Numpy' library is required for array management.]

The output will also contain the color bar, xlabel, ylabel, x-axis limit, and y-axis limit as per the above-mentioned commands.

Output:

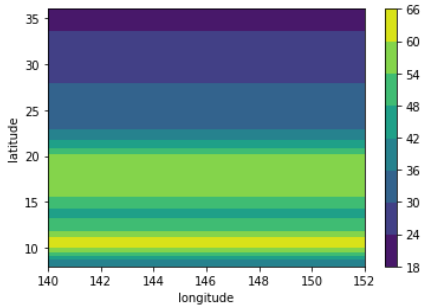


Figure 9.16. Contour plot

Three dimensions (3D) contour plots can be created by importing 'mplot3d' module from 'mpl_toolkits' library and using 'contour3D()' function. We also need 'numpy' and 'matplotlib.pyplot' libraries. For 3D plotting, we also require to use the 'projection'

argument in ‘plt.axes()’ function. Projection should be entered as ‘3d’ in ‘plt.axes()’ function:

```
from mpl_toolkits import mplot3d

import numpy as np

import matplotlib.pyplot as plt

fig = plt.figure()

ax  = plt.axes(projection='3d')

ax.contour3D(lon, lat, MSW_ms, 50, cmap='viridis')

ax.set_xlabel('longitude')

ax.set_ylabel('latitude')

ax.set_zlabel('Maximum sustained wind speed (m/s)')

plt.show()
```

[N.B. Here, the command for 3D contour plotting can be expressed as:

```
ax.contour3D(lon, lat, MSW_ms, 50, cmap='viridis')
```

Where, lon, lat, MSW_ms = X, Y, Z.]

Output:

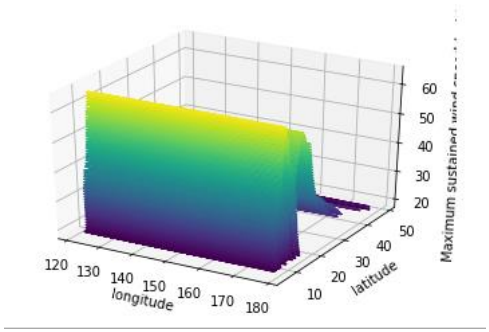


Figure 9.17. 3D contour plot

1. Spatial mapping in Python with Cartopy, Xarray, and NetCDF4

We need to install netCDF4, cartopy, and xarray in **Anaconda** by imitating the given steps:

Install netcdf4:

Step 1: run Anaconda as Administrator

Step 2: we need to use the following command if we use an anaconda

```
conda install -c anaconda netcdf4
```

Or we need to use the following command if we use python

```
pip install netCDF4
```

Install cartopy:

Step 1: run Anaconda as Administrator

Step 2: we need to use the following command if we use an anaconda

```
conda install -c conda-forge cartopy
```

Install xarray:

Step 1: run Anaconda as Administrator

Step 2: we need to use the following command if we use an anaconda

```
conda install -c anaconda xarray
```

After installation the abovementioned packages, we need to open the Jupyter Notebook and follow the given steps:

- The following commands are for importing necessary libraries:

```
from netCDF4 import Dataset
import xarray as xr
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import netcdf
import cartopy.crs as ccrs
```

[N.B. ‘Dataset’ from netcdf4 and ‘xarray’ library will be required to read the NetCDF (.nc) data and variables. ‘Pandas’ library should be imported for excel, .csv files, and similar types of data handling and management. ‘Pyplot’ module of the ‘Matplotlib’ library is required for data visualization and plotting. ‘Numpy’ should be imported for array management. The ‘Netcdf’ module from ‘scipy.io’ library can also be used for NetCDF (.nc file) data management, e.g. masking the data. We will require ‘cartopy.crs’ library for mapping with appropriate projection system and geospatial analyses plotting, where crs means ‘Coordinate Reference Systems’.]

1.1. How to know about the data

We need to draw maps if we want to show the change of ‘Z’ value as functions of ‘X’ and ‘Y’ variables. In terms of geographical position/ location, ‘X’ is longitude (range - 90°S to 90°N), and ‘Y’ is latitude (range - 180°W to 180°E). In the world map, our Earth is divided into several grids. We can find any location/place by simply using the latitude and longitude of that place.

We can use ‘Dataset(r‘file.nc’)’ function to read our dataset ‘**precipitation.nc**’:

```
data = Dataset(r'precipitation.nc')

print(data.variables.keys())
```

The ‘Dataset’ module read the dataset as a dictionary, consisting of keys and variables. Hence, we can print the variables’ names by the above-mentioned codes.

The output will print the short names for the variables as the keys of the dictionary.

Output:

```
dict_keys(['gwgt', 'lat', 'lon', 'time', 'prc'])
```

We can use a loop to print the details about all the variables of the dataset:

```
for var in data.variables.values():

    print(var)
```

Output:

```
<class 'netCDF4._netCDF4.Variable'>
float32 gwgt(lat)
    long_name: gaussian weights
    short_name: gwgts
    units: dimensionless
unlimited dimensions:
current shape = (64,)
filling on, default _FillValue of 9.969209968386869e+36 used
<class 'netCDF4._netCDF4.Variable'>
float32 lat(lat)
    short_name: lat
    long_name: latitude
    units: degrees_north
unlimited dimensions:
current shape = (64,)
filling on, default _FillValue of 9.969209968386869e+36 used
<class 'netCDF4._netCDF4.Variable'>
float32 lon(lon)
    short_name: lon
    long_name: longitude
    units: degrees_east
unlimited dimensions:
current shape = (128,)
filling on, default _FillValue of 9.969209968386869e+36 used
<class 'netCDF4._netCDF4.Variable'>
int32 time(time)
    long_name: Year-Month
    short_name: YRMO
    units: yyyymm
    _FillValue: -999
unlimited dimensions:
current shape = (216,)
filling on
<class 'netCDF4._netCDF4.Variable'>
float32 prc(time, lat, lon)
    short_name: PRC
    long_name: Precipitation
    units: mm/day
    _FillValue: -999.0
unlimited dimensions:
current shape = (216, 64, 128)
filling on
```

[N.B. The dataset can be described as follows:

- The long name for ‘gwgt’ is ‘gaussian weights’ with no units and a dimension of 64
- The long name for ‘lat’ is ‘latitude’ with ‘degrees_north’ units and a dimension of 64

- The long name for ‘lon’ is ‘longitude’ with ‘degrees_east’ units and a dimension of 128
- The long name for ‘time’ is ‘Year-Month’ with ‘yyyymm’ units and a dimension of 216
- The long name for ‘prc’ is ‘Precipitation’ with ‘mm/day’ units and three dimensions of (time, lat, lon) or (216, 64, 128)

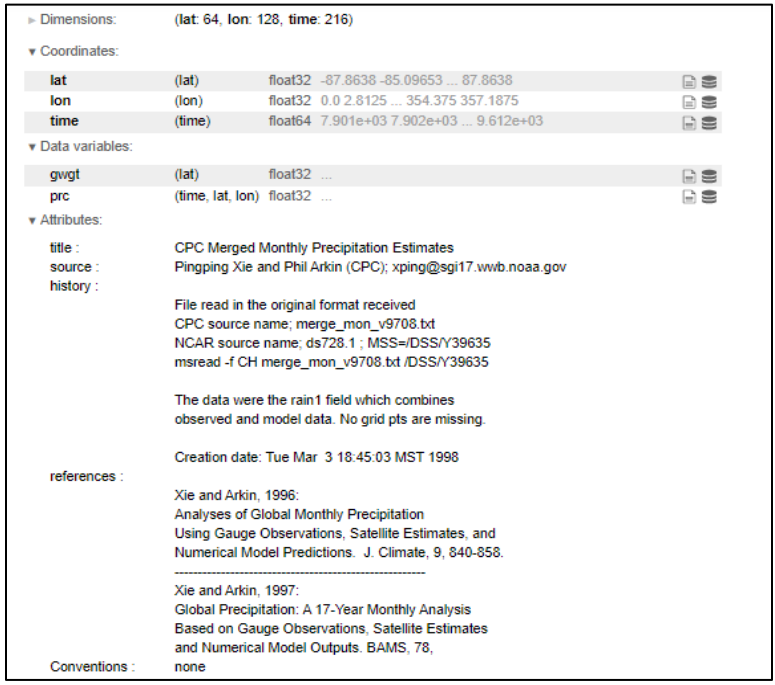
Here, ‘Precipitation (prc)’ is the main variable which consists of three-dimensional data with time (18 years), latitude and longitude.]

We can also use ‘xarray’ library to print our dataset by using the ‘open_dataset()’ function:

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

ds
```

Output:



1.2. How to know about the variables

We can also get the details about the specific variables of a dataset as follows:

```
data.variables['prc']
```

The output will give details information about only one ‘prc’ variable of the ‘precipitation.nc’ dataset.

Output:

```
<class 'netCDF4._netCDF4.Variable'>
float32 prc(time, lat, lon)
    short_name: PRC
    long_name: Precipitation
    units: mm/day
    _FillValue: -999.0
unlimited dimensions:
current shape = (216, 64, 128)
filling on
```

1.3. How to know about the minimum and maximum latitude and longitude for mapping

We need to know the maximum minimum latitude and longitude of the variables in the dataset before further analysis. Hence, we can follow the given codes:

```
print(“min_lat  = ”, min(data.variables[‘lat’]))

print(“max_lat  = ”, max(data.variables[‘lat’]))

print(“min_lon  = ”, min(data.variables[‘lon’]))

print(“max_lon  = ”, max(data.variables[‘lon’]))
```

Output:

```
min_lat  =  -87.8638
max_lat  =  87.8638
min_lon  =  0.0
max_lon  =  357.1875
```

[N.B. The dataset covers the global data where latitude limits are from -87.86 to + 87.86 (around 90°) and longitude ranges from 0 to 357.19 (around 360°). Hence, this represents a global dataset.]

We can access all the data from each variable using indexing and slicing operators as ‘[:]’. Now, we need to convert the ‘prc’ variable into two-dimension (2D) over time for mapping. Therefore, we will average our precipitation data over the ‘time’ variable (axis = 0) for 2D conversion:

```
lat  = data.variables[‘lat’][:]
```

```
lon  = data.variables['lon'][:]

time = data.variables['time'][:]

prc  = data.variables['prc'][:]

mean_rainfall = np.mean(prc, axis=0)

print(mean_rainfall)
```

Output:

```
[[0.328615   0.32649407 0.32465214 ... 0.3369908  0.333855   0.33105692]
 [0.2591474  0.26262572 0.26580238 ... 0.24838836 0.25187734 0.25541523]
 [0.39385906 0.39131266 0.38667652 ... 0.3910227  0.39348662 0.3945333 ]
 ...
 [0.57567424 0.5975326  0.6245758  ... 0.55330473 0.5529246  0.56047904]
 [0.64617604 0.65975326 0.67477953 ... 0.61600125 0.6242001  0.6342811 ]
 [0.5605251  0.57392156 0.5873435  ... 0.52158374 0.5343335  0.5472972 ]]
```

1.4. Print contour map

Contour plots can be used when we have three-dimensional data (or more than two-dimensional data) e.g., Z = f(X, Y), where the Z value changes as a function of two inputs: X and Y. In our 'precipitation.nc' dataset, we can plot a contour map of 'prc' as functions of 'lon' and 'lat' using 'contourf()' function:

```
plt.contourf(lon, lat, mean_rainfall)

plt.xlim(60,100)

plt.ylim(5,30)

plt.xlabel('Longitude')

plt.ylabel('Latitude')
```

[N.B. The commands can be expressed as follows:

plt.contourf(X, Y, Z), where X, Y, Z – variables

plt.xlim(lower,upper), where xllim – x-axis range

plt.ylim(lower,upper), where ylim – y-axis range

plt.xlabel('label'), where xlabel – label for the x-axis

plt.ylabel(‘label’) where ylabel – label for the y-axis]

Output:

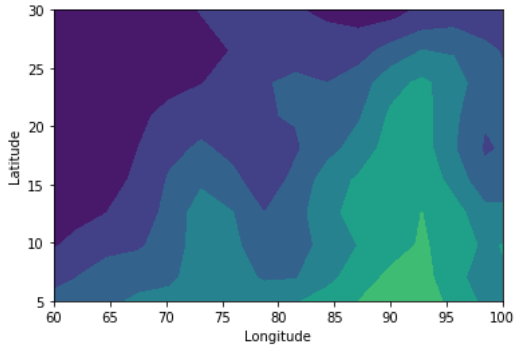


Figure 10.1. Contour map

1.5. Understanding the transform and projection keywords

In the above map, we need to add a shapefile for a geographical location to add the area of a place. By default, every programming language contains the world’s shapefile. Hence, we can read as well as mask the shapefile for our dataset using the ‘netcdf_file()’ function. We have to access all the variables using indexing and slicing operators as ‘[:]’. [0, : , :] for ‘prc’ variables means we want to access only the first row’s data for the ‘time’ dimension, but all data for latitude and longitude variables. Then, we have to project our dataset using ‘axes()’ function. Finally, we can plot a contour map of ‘prc’ as functions of ‘lon’ and ‘lat’ using ‘contourf()’ function along with setting the colour map and transformation, and adding the coastlines, and gridlines in the global map:

```
dataset = netcdf.netcdf_file('precipitation.nc', maskandscale = True, mmap = False)

prc  = dataset.variables['prc'][0, :, :]

lats = dataset.variables['lat'][:]

lons = dataset.variables['lon'][:]

ax   = plt.axes(projection=ccrs.PlateCarree())

plt.contourf(lons, lats, prc, cmap = 'YlOrRd', transform=ccrs.PlateCarree())

ax.coastlines()
```

```
ax.gridlines(draw_labels=True)

plt.show()
```

The output will give us a global map.

Output:

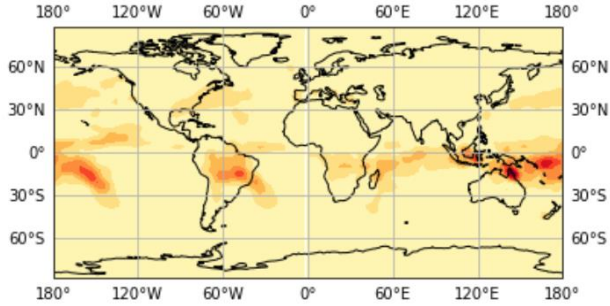


Figure 10.2. Global map

We can get a regional map by using 'set_extent()' function along with the previously mentioned codes as follows:

```
dataset = netcdf.netcdf_file('precipitation.nc', maskandscale = True, mmap =

      False)

prc  = dataset.variables['prc'][0, :, :]

lats = dataset.variables['lat'][:]

lons = dataset.variables['lon'][:]

ax   = plt.axes(projection=ccrs.PlateCarree())

plt.contourf(lons, lats, prc, cmap = 'YlOrRd', transform=ccrs.PlateCarree())

ax.set_extent([30, 170, -30, 30])

ax.coastlines()

ax.gridlines(draw_labels=True)
```

```
plt.show()
```

[N.B. The commands for the 'set_extent()' function to get a regional map can be expressed as follows:

```
ax.set_extent([x0, x1, y0, y1])
```

Here, x0 and x1 are the lower and upper limit of longitude, and yo, y1 are the lower and upper limit of the latitude range.]

The regional map will be visualized in the output.

Output:

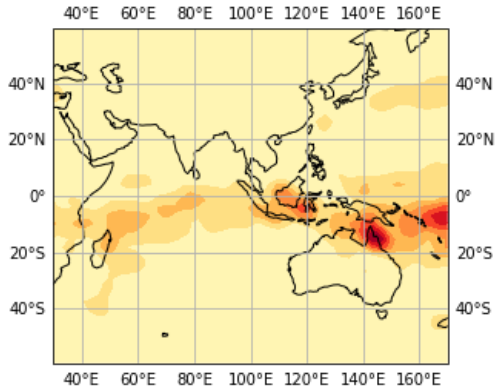


Figure 10.3. Regional map

[N.B. The following websites can provide more information regarding transform and projection keywords with lists:

- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
- https://scitools.org.uk/cartopy/docs/v0.15/crs/projections.html]

1.6. Adding Features to the Map

We can add some extra features of the previously prepared map by using 'cartopy' as well as 'xarray' libraries. We can add color bar, ocean, land, lakes, and river features and remove the right and top grid labels using 'cartopy' library as follows:

```
import cartopy.feature

ax   = plt.axes(projection=ccrs.PlateCarree())
```

```
plot = plt.contourf(lons, lats, prc, cmap = 'YlOrRd', transform = ccrs.PlateCarree())

plt.colorbar(plot, ax=ax, shrink=0.8)

ax.set_extent([30, 170, -30, 30])

ax.coastlines()

gl = ax.gridlines(draw_labels=True)

gl.top_labels   = False

gl.right_labels = False

ax.add_feature(cartopy.feature.OCEAN)

ax.add_feature(cartopy.feature.LAND, edgecolor='black')

ax.add_feature(cartopy.feature.LAKES, edgecolor='black')

ax.add_feature(cartopy.feature.RIVERS)

plt.show()
```

[N.B. Here, the command for extra features like color bar and gridline label can be expressed as:

```
plt.colorbar(plot, ax=ax, shrink=0.8)
gl.top_labels    = False #for removing top gridline labels
gl.right_labels = False #for removing top gridline labels ]
```

The output of the regional map with color bar, ocean, land, lakes, rivers features and no top-right gridline labels will be as follows.

Output:

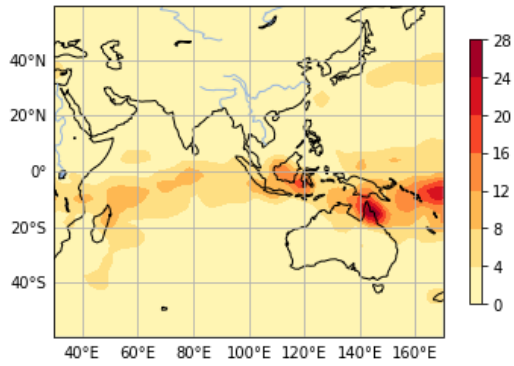


Figure 10.4. Regional map with modifications using cartopy

The same features can be added by using 'xarray' library with the following similar codes:

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

prc  = ds.prc[0, : , : ]

ax1   = plt.axes(projection = ccrs.PlateCarree())

ax1.set_extent([30, 170, -30, 30])

ax1.coastlines()

gl = ax1.gridlines(draw_labels=True)

gl.top_labels   = False

gl.right_labels = False

gl.right_labels = False

prc.plot(ax=ax1, cmap = 'YlOrRd', transform = ccrs.PlateCarree(), cbar_kwargs =
        {'shrink': 0.8})

plt.title('')

plt.show()
```

The output of the regional map with color bar, coastline features and no top-right gridline labels using 'xarray' library will be as follows.

Output:

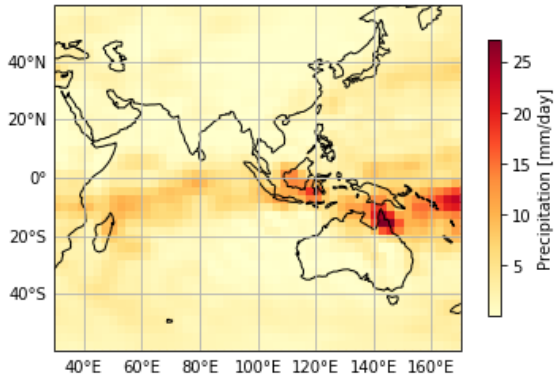


Figure 10.5. Regional map with modifications using xarray library

[N.B. Here, the commands for extra features like color bar and removing gridline labels are quite similar to the codes used in 'cartopy' library to get those features:

```
prc.plot(ax=ax1, cmap = 'YlOrRd', transform=ccrs.PlateCarree(), cbar_kwargs={'shrink': 0.8}
```

The following websites can provide more information regarding adding features while spatial mapping in python:

- https://rabernat.github.io/research_computing_2018/maps-with-cartopy.html]