Python book

Manish Datt

Table of contents

# Preface

Start your coding journey with Python Book! This book is ideal for beginners to learn to program. Python is a modern high-level programming language that has applications in diverse fields.

To learn more about Bioinformatics learning resources, please visit [bioinfo.guru](https://bioinfo.guru).

For any query or feedback related to Python Book, please reach out to the author at manish@bioinfo.guru

# Introduction

|  |
| --- |
| **Steve Jobs**  “Everyone in this country should learn how to program because it teaches you how to think” |

## Why learn programming?

This is a frequently asked question that is why I should learn a programming language. There are multiple merits of learning the basic framework of computer programs. First, learning a programming language helps us develop logical skills, that’s because programming requires breaking down a problem and approaching a solution in a logical manner. All the computer algorithms are based on the idea that – understand the problem, break it down into different steps, and then go about solving the problem step by step. In the modern world everything is based on computers and having an ability to communicate with computers (via programming language) would help us understand the basis of functioning of any computer program. We will be able to “sense” the workflow of a program and to draw a relationship between input(s) and output(s).

## Why learn Python?

Another frequently asked question is why should I learn Python programming Python is the modern programming language that is being actively developed. It offers all the features expected out of a high and programming language that too with a rather flat learning curve. The coding syntax is pretty much like writing in plain English so that it makes it easier for a novice to learn this programming language. This advantage is particularly applicable for researchers in the field of Biology because, conventionally, students of biology have little or no exposure to computer related subjects. So, to get them started with programming, Python is an ideal choice. There are of course other advantages as well, like Python is an object-oriented programming language. It is an interpreted language i.e. the code is executed line by line which helps a new coder to identify and debug errors somewhat easily. Most importantly, python offers plenty of libraries to add additional functionality specific to a particular domain. For example, libraries like NumPy and Matplotlib can be used for numerical analysis and data visualization, respectively. Active development and a large user community allows exchange of ideas and speed up troubleshooting tasks.

## I’m a biologist, why should I learn to code?

Modern Biology research is highly data intensive. Platform technologies are generating data like never before; and to parse the data we need computers. To use computers innovatively and productively, knowledge of programming is a must. Further, there are several software available for bioinformatics research; understanding programming concepts would enable us to interface with these tools in a much better and efficient manner than otherwise. To be a skilled computational biologist we need to understand the workflow that connects – concepts to algorithms to coding. One need not be an expert in all these domains but must have a basic understanding of this workflow. Once you have a vision of the big picture and you are an expert in at least one of these components then you can always collaborate with other experts to do wonderful things together.

In Bioinformatics, particularly in the recent past, a lot of tools have been developed using python, e.g. one of the most popular structure visualization programs PyMol is developed using Python. Similarly, there are many other libraries that facilitate different kinds of analysis using biological data. Considering all these factors, the ability of code in python is an asset for biologists doing data intensive research.

# 1. Setup

## 1.1 Python distributions

The first thing that you need to do is to install Python. There are various distributions available online one of the such option is to download from the python offical site. [Anaconda](https://www.anaconda.com/products/distribution) is another popular python distribution. Once you have download the executable file the installation is straight forward process.

## 1.2 Integrated Development Environments

After python installation the next thing you need is a code editor. Although any text editor (such as Notepad++/vi/TextEdit) can serve this purpose but it is highly recommend to install atleat one IDE (Interactive Development Environment). The Anaconda distribution comes bundled with Jupyter (an IDE). VS Code (Visual Studio Code) is an other popular IDE among software deveplopers. The IDEs have several useful features such as syntax highlighting, code auto-completion, options to install plugin, etc. that elevate the coding experience. The figure below shows the interface for Jupyter and VS Code.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | (a) Jupyter | |  |

|  |  |
| --- | --- |
| |  | | --- | | (b) VS Code | |

Figure 1.1: Integrated Development Environments (IDEs)

In addition to these IDEs, Google Collaboratory is an online Jupyter notebook that can be used to execute python code in the cloud.

Throughout this book, the python code is shown as a code block with grey background, e.g.

print("Hello World!")

|  |
| --- |
| Quick tip |
| When you hover the mouse pointer over a code cell, there will be an option on the right to copy the contents of that cell. |

## 1.3 Useful links

| Name | Website |
| --- | --- |
| Python website | <https://python.org> |
| Anaconda | <https://www.anaconda.com/products/individual> |
| Google Colab | <https://research.google.com/colaboratory/> |
| VS Code | <https://code.visualstudio.com/> |

# 2. Variables

In programming variables refers to a “name” for a value. This “name” can have different values over the course of program and hence it is referred as a variable. The concept of variable in programming is similar to some of terms used in natural languages to refer to different things. For instance, in a classroom there are students with different *names*. Here the word *name* is a variable which has differnt *value* for each student i.e. their name.

a = 2

The above statement initializes a variable named ‘a’ with a numeric value of 2 and it is refered to as an **Assignment Statement**. Similarly, the statement below initializes a variable with name ‘test’ and assigns a string (or a text) value of ‘Happy’

test = 'Happy'

We can print the value of a variable using the print command i.e.

print(a)  
print(test)

2  
Happy

The value assigned to a variable can be changed within the program. This feautre make variable very useful and are extensively used in progamming. A variable, for example, can be used to store user input.

Each variable in Python has 6 properties associated with it. For now only first three are required since the last three are for advanced programmers. These properties are 1. Name 2. Value 3. Type 4. Scope 5. Life time 6. Memory location

The first two properties i.e. name and value we have already seen above. Type for a variable refers to it’s data type i.e. whether the variable is an integer, float, string, boolean, etc.

In Python, the type for a variable is decided at the time of initialization. This behaviour is different from languages like C where the type of the variable has to be declared explicitly. In the example above, we can change the variable ‘a’ to a non numeric value. To check the type for a particular variable, type() function can be used as follows.

x = 7  
y = 7.0  
print(type(x))  
print(type(y))

<class 'int'>  
<class 'float'>

Variable names are case sensitive, e.g. in the code below there are four different variables such that their name differ only in the case used for the letters.

bat = 4  
BAT = 5  
Bat = 6  
bAT = 7  
print(bat, BAT, Bat, bAT)

4 5 6 7

## 2.1 Operators

An operator is a special symbol that instructs the compiler/interpreter to execute specific mathematical or logical operations. These symbols *operate* on set of variable or scaler or both. Operand refers to the value that operator operates on. E.g., in the equation 2+3 ‘+’ is an operator and 2 and 3 are operands.

In Python there are 6 different classes of operators. These are: 1. Arithmetic operators 2. Comparison operators 3. Logical operators 4. Assignment operators 5. Bitwise operators 6. Special operators

Let’s understand each of these classes through some examples

## 2.2 Arithmetic operators

As the name suggests this class of operators are used to perform mathematical operation on the operands. Following is the list of arithemetic operators available in Python.

| Operator Symbol | Meaning | Example |
| --- | --- | --- |
| + | Addition | 2 + 2 = 4 |
| - | Subtraction | 5 - 2 = 3 |
| \* | Multiplication | 2 \* 3 = 6 |
| / | Division | 4 / 2 = 2 |
| % | Modulus | 4 % 2 = 0 |
| // | Floor division | 5 // 2 = 2 |
| \*\* | Exponent | 2 \*\* 3 = 8 |

x = 4  
y = 5  
print (x+y+2)  
print (x-y+2)

11  
1

## 2.3 Comparison operator

These operators are used to compare operands and the output from their operation is a boolean value i.e. either True or False or equivalently 1 or 0.

Following is the list of arithemetic operators available in Python.

| Operator Symbol | Meaning | Example |
| --- | --- | --- |
| > | Greater than | 3 > 2 is True |
| < | Less than | 3 < 2 is False |
| == | Equal to | 2 == 3 is False |
| != | Not equal to | 2 != 3 is True |
| >= | Greater than or equal to | 4 >= 2 is True |
| <= | Less than or equal to | 5 <= 2 is False |

x = 4  
y = 5  
print (x+y > 10)

False

## 2.4 Logical operators

There are 3 logical operators and, or, and not.

| Operator Symbol | Meaning |
| --- | --- |
| and | True if both operands are true |
| or | True if either of the operands is true |
| not | True if operands are false |

x = True  
y = False  
print (x and y)  
print (x or y)  
print (not x)

False  
True  
False

x=5  
y=6  
z=7  
print (x<y and y<z)

True

## 2.5 Assignment operators

Operators in this class are used to assign values to variables. The syntax for using these operators is variable name followed by operator followed by the value that is to be assigned. Some of the assignment operator behave equivalent to the mathematical operation performed by arithmetic operators.

Following is the list of assignment operators available in Python.

| Operator Symbol | Meaning | Example |
| --- | --- | --- |
| = | Assign value on the right to variable on the left | x = 5 |
| += | Increment value of the variable by value on the right | x += 5 (x = x + 5) |
| -= | Decrement value of the variable by value on the right | x -= 5 (x = x - 5) |
| \*= | Multiple value of the variable by value on the right | x \*= 5 (x = x \* 5) |
| /= | Divide value of the variable by value on the right | x /= 5 (x = x / 5) |
| %= | Modulus for the value of the variable by value on the right | x %= 5 (x = x % 5) |
| //= | Floor division of value of the variable by value on the right | x //= 5 (x = x // 5) |
| \*\*= | Exponent value of the variable by value on the right | x \*\*= 5 (x = x \*\* 5) |

Quiz: What would be the output of the following code x = 6 x \*\*= 2 print(x)

36

Quiz: What would be the output of the following code x = False print(type(x))

x = False  
print(type(x))  
  
#output   
#<class 'bool'>

# 3. Strings

In programming string refers to a sequence of characters that can act as a variable or constant. This is the most popular data type in Python. In fact the increasingly strong prevelance of Python in Bioinformatics is primarily due to its ability to easily perform different operations of strings. A string variable can be assigned a value using either single or double quotes.

x = 'String A'   
var2 = "Another variable"  
print(x)  
print(var2)

String A  
Another variable

To check the data type of a variable, we can use the type function. The get number of character in a string variable use the len function. Note that a blank space is also considered as a character.

print(type(x))  
print(len(var2))

<class 'str'>  
16

## 3.1 String concatenation

The process of joining two or more things is called as concatenation. The arithmetic operators + and \* can be used directly with strings to concatenate (addition) or repeat (multiplication). This process of giving additional functions to operator (beyond their existing functions) is called as operator overloading. E.g. the plus (+) operator is used for addition given that the operands are integers. However, if the operands are string then it act as a concatenation operator instead of addition operator.

# The plus (+) operator with two numbers  
2+3

5

var1 = "Hello"  
var2 = "World!"  
print (var1+var2)

HelloWorld!

In the case of asteriks (\*) operator, which is used to multiple two numbers, when the operands are a string (*s*) and a number (*n*) the output is *s* repeated *n* times. This behaviour is similar to muliplication of two numbers. For instance, let say we want to multiple 5 by 3 (5\*3). This multiplication can also be represented as sum of 5 three times i.e. 5+5+5. So, when we use a string (*s*) and a number (*n*) as operands for \* operator we get *s*+*s*+*s*…(*n* times).

var3 = var1\*3  
print(var3)

HelloHelloHello

## 3.2 Slice of a string

Slice is another very useful operator that can be used to manipulate strings. The slice operator [] gives the character within the start and end positions separated by a colon. The numbering of characters within a string start from 0. Note that the start position character is included in the output but the end position character is not. Slicing effectively return the substring of a given string. The general syntax for slicing a string is as follows:

string[start:end] string[start:] string[:end] string[start:end:step]

Let’s see some examples to get a better understanding to the slice operation.

var4 = "ABCDEFG"  
print(var4)  
print(var4[1:5])

ABCDEFG  
BCDE

In case no value is specified before or after the colon then the slicing would occur from begining or till end respectively.

print(var3)  
print(var3[:7])  
print(var3[3:])

HelloHelloHello  
HelloHe  
loHelloHello

The **step** part in the slice operator specific the number of steps to take when going from the start position to the end position. The default step size is 1. We can change the default value by specifying the step parameter within the slice command.

print(var3)  
print(var3[2::2])

HelloHelloHello  
loelHlo

Quiz: Write a command that outputs ‘HHH’ given a string ‘HelloHelloHello’.

var3 = "HelloHelloHello"  
print(var3[::5])

## 3.3 String comparison

One of the frequently required tasks in programming is string comparison. In Python comparison operator can be used to compare two strings. The == (two equal symbols without space) is the comparison operator. The output of comparison is a boolean value i.e. either True or False. String comparison is case sensitive.

var1 = 'Hello'  
var2 = "Hello"  
var3 = 'Hi'  
print(var1 == var2)  
print(var1 == var3)  
print(var1 == "hello")

True  
False  
False

## 3.4 Splitting string

Sometimes there is a need to split a string based on certain delimiters, the split function is designed for that task. Python String types have split function associated with them that return a *list* of elements after splitting the string as per the delimiter. The default delimiter is blank space.

s1 = "This is a sentence."  
words1 = s1.split()  
print(words1)  
  
#split with comma as a delimiter  
s2 = 'This is an another sentence, a longer one.'  
words2 = s2.split(",")  
print(words2)

['This', 'is', 'a', 'sentence.']  
['This is an another sentence', ' a longer one.']

Quiz: What would be the output if we split s2 using “is” as a delimiter.

s2 = 'This is an another sentence, a longer one.'  
print(s2.split("is"))  
  
##Output would be a list with three elements:  
##['Th', ' ', ' an another sentence, a longer one.']

## 3.5 String functions

Python strings have several methods to work with string objects. Below are examples of some of the functions available is class ‘str’. These methods acts on the string and returns a new string after doing the required manipulations. For additional functions, please refer to the python documentation.

s1 = "Apple"  
  
String function Output  
————————————————————————————————  
s1.upper() APPLE  
s1.lower() apple  
s1.startswith("a") False  
s1.startswith("A") True  
s1.index("l") 3  
s1.replace("e","es") Apples

## 3.6 Doc strings

To declare a variable whose value is a long string that spans multiple lines tripple quotes can be used. All white spaces such as tabs and newline are considered part of the string. These types of strings are generally used for documentation purposes e.g. writing help text for custom functions.

var4 = """This is an example of  
a long string that spans   
three lines."""  
print (var4)

This is an example of  
a long string that spans   
three lines.

## 3.7 Strings – key characteristics

* **A string variable stores text data**

A string variable can store characters including white spaces (space, tab, newline). A string variable is an object of class ‘str’. To initialize a string variable, single or double quotes can be used.

* **A string variable is immutable**

In Python the strings are immutable i.e. their value cannot be changed once it has been assigned. The values can however be reassigned. The value of a string variable can change but the data contained within a variable can`t be changed.

* **A strings variable is a list**

A string variable is a also a list i.e. a collection of characters. We can iterate through characters in a string just like we can iterate through any list. Unlike lists, however, characters cannot be appended to a string because strings are a immutable data type.

# 4. Numbers

Python has a couple of different options for data types when it comes to working with numbers. Numbers without and with demical points are refered to as int (integers) and float (floating point). So, depending upon how we initialize a variable, its data type would be assigned accordingly. Both these data types can have negative values. The int and float functions can be used to explicity assign the corresponding data types. Note, when assigning and int data type to a number with decimal value, only the interger part is retained. For working with complex numbers there is an additional data type complex.

x = 5   
y = 5.0  
print(type(x))  
print(type(y))  
a = float(x)  
b = int(y)  
print(a)  
print(b)

<class 'int'>  
<class 'float'>  
5.0  
5

A numeric variable can also be initiatlized using scientific notation i.e. e notation. The e can be both lower or upper case. Variables declared with a scientific notation are always float.

z = 5e3  
print(z)

5000.0

The data type for the output of an arithmetic operator would depend on the data type of the operands. If both the operands are int then the output would also be int. An exception to this is / (division) where the data type for the output is always float irrespective of the data type of the operands. Also, if at least one of the operand is float then the data type for the output would be float.

print(type(4\*2))  
print(type(4/2))

<class 'int'>  
<class 'float'>

## 4.1 Formatted print

To print a floating point number with a desired number of decimal places we can use the formatted print option. The %f format specifier can be used to indicate the required number of decimal places for the output. Here, we also have an option to include the sign (+) in case of positive number. In addition to this, the round function can also be used which takes two arguments – the floating point number and the required number of decimal places.

n = 1234.5678  
  
Formatted print Output  
————————————————————————————————  
print("%.2f" % n) 1234.57  
print("%+.2f" % n) +1234.57  
print("%.2e" % n) 1.23e+03  
print("%.d" % n) 1234

## 4.2 f-strings

From python 3.6 onwards, formatting of variable, calculations of mathematical expression etc. can be easily done within the print statement. These string starts with f. The variables inside the f-strings are enclosed within curly braces {}. The formatting is similar to as we described above, the code is relatively cleaner and easy to follow.

a = -4  
b = 5.0  
print(f'The product of {a} and {b} is {a\*b}')  
print(f'The product of {a} and {b} is {a\*b:0.3f}')

The product of -4 and 5.0 is -20.0  
The product of -4 and 5.0 is -20.000

## 4.3 Complex numbers

Complex numbers can be initialized using the complex function which takes either one or two arguments, depending upon the nature of argument: - a complex number is specified directly - the real and imaginary parts are specified as separate arguments.

c = complex(3+6j)  
z = complex(5,2)  
print(c, z)  
print(f"The real component of z is {z.real}")  
print(f"The imaginary component of z is {z.imag}")  
print(f"The conjugate of z is {z.conjugate()}")

(3+6j) (5+2j)  
The real component of z is 5.0  
The imaginary component of z is 2.0  
The conjugate of z is (5-2j)

# 5. Lists

A collection of items, in python, is refered to as a list. A list can have elements of same data type or multiple data types. A list is denoted by square brackets ([]). The indexing of elements in a list starts from zero. To access an element by its index we can specify index in square brackets after the list name. Slicing of lists can be performed as well.

fruits = ["apple", "banana"]  
print(type(fruits))  
print(fruits)  
print(fruits[1])

<class 'list'>  
['apple', 'banana']  
banana

numbers = [10,20,30,40,50]  
print(numbers)  
print(numbers[1:4]) # slicing

[10, 20, 30, 40, 50]  
[20, 30, 40]

## 5.1 Modifing contents of a list

A list is a mutable data type i.e. the contents of the collection can be changed. The append and extend functions adds an element or a collection of elements to a list. To add an element at a specified position within the list, the insert function can be used.

fruits.append("mango")  
print(fruits)  
print(len(fruits))

['apple', 'banana', 'mango']  
3

fruits = ["apple", "banana"]  
fruits2 = ["pineapple","cherry"]  
fruits.append(fruits2) #adding a list using append  
print(fruits)  
print(len(fruits))  
  
fruits = ["apple", "banana"]  
fruits2 = ["pineapple","cherry"]  
fruits.extend(fruits2) #adding a list using extend  
print(fruits)  
print(len(fruits))

['apple', 'banana', ['pineapple', 'cherry']]  
3  
['apple', 'banana', 'pineapple', 'cherry']  
4

fruits.insert(1,"grapes")  
print(fruits)

['apple', 'grapes', 'banana', 'pineapple', 'cherry']

# slicing  
print(fruits[1:4])

['grapes', 'banana', 'pineapple']

To remove the last element from a list, pop function can be used. By default, it returns a the last element and the original list is shortened by one element. The optional positional argument to specify the index of the element that is to be “popped”.

last\_fruit = fruits.pop()  
print(fruits)  
print(last\_fruit)

['apple', 'banana', 'pineapple']  
cherry

fruits = ['apple', 'banana', 'pineapple', 'cherry']  
popped\_fruit = fruits.pop(2) # pop the element at index 2.  
print(popped\_fruit)  
print(fruits)

pineapple  
['apple', 'banana', 'cherry']

To remove an element by its name use the remove function. Note that if a list has duplicate elements then the remove function will delete only the first occurance of that element.

fruits.remove("banana")  
print(fruits)

['apple', 'pineapple']

## 5.2 List functions

There are some useful functions available to manupulate lists. These functions act in place i.e. these functions do not return anything and just modifies the original list.

fruits = ['apple', 'banana', 'mango', 'pineapple', 'cherry', 'banana']  
print(fruits.count("banana"))  
fruits.reverse()  
print(fruits)  
fruits.sort()  
print(fruits)  
fruits.sort(reverse=True)  
print(fruits)

2  
['banana', 'cherry', 'pineapple', 'mango', 'banana', 'apple']  
['apple', 'banana', 'banana', 'cherry', 'mango', 'pineapple']  
['pineapple', 'mango', 'cherry', 'banana', 'banana', 'apple']

Quiz: Given a list nums=[1,2,3,4,5]. Write a code to print 4.

nums = [1,2,3,4,5]  
print(nums[3])

# 6. Conditional Statements

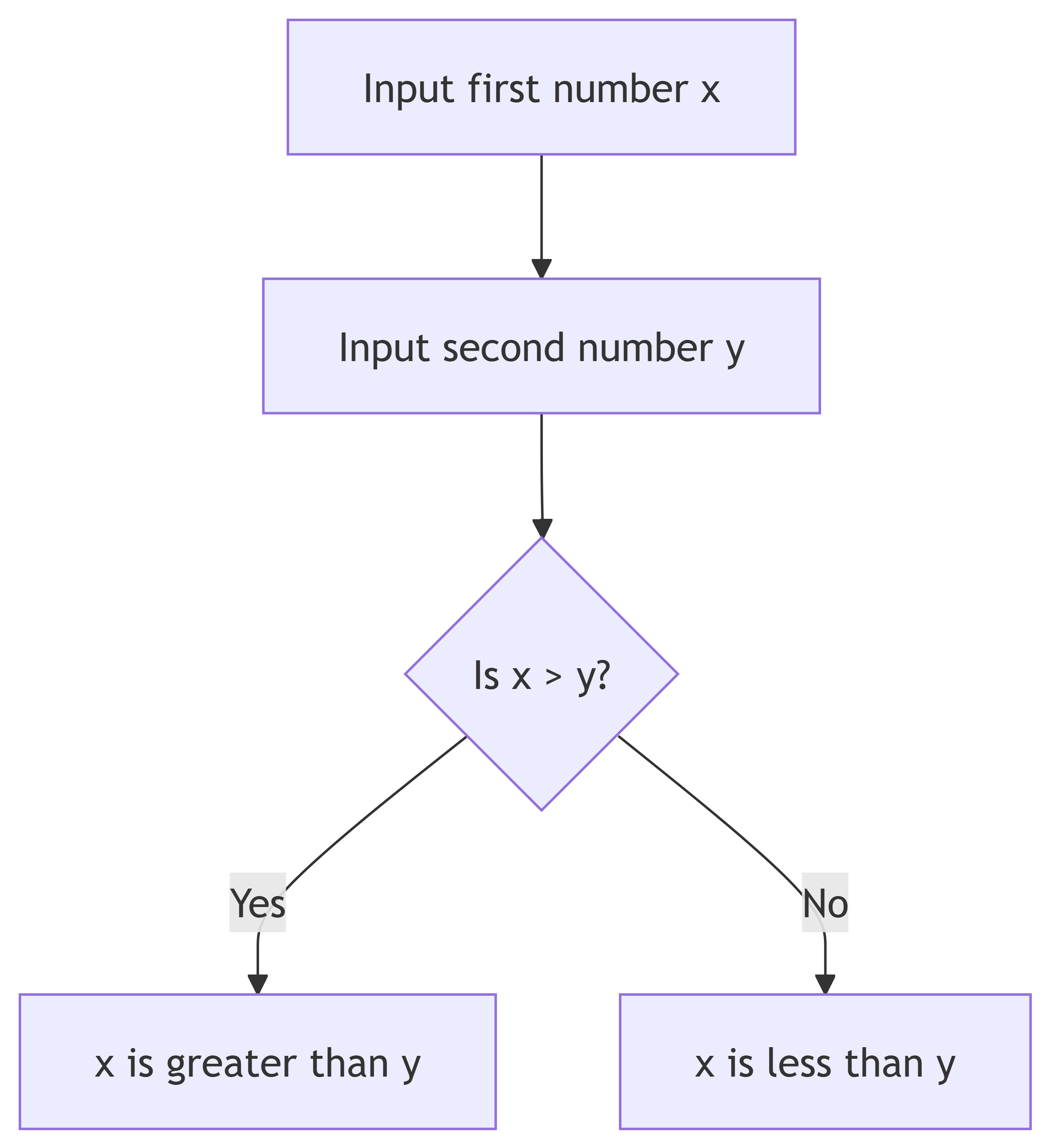
Many a times in the life cycle of a program there are instances when you would like to run a certain code based on whether a particular condition is met or not. A real-world example of this would be screening of content based on the user’s age. If, e.g., the user is 18+ then all the content is accessible to the user while if the user is below 18 then certain content is restricted. These kind of testing of condition(s) and then executing the specific code accordingly is facilitated by the conditional statements.

## 6.1 if and else statements

To test a particular condition, if statement is used with some condition that we need to check. This kind of checking generally involves a comparison operator such that the output is always Boolean (true/false). In case the output of the comparison is true the code block following the if statement gets executed. On the other hand, if the condition is false then the code specified under the else block gets executed. Note that there is no testing of condition in case of else statement since the else directly means when the if condition is false.

When using conditional statements, there are two syntactical points that you need to keep in mind:

* The if and the else statements ends with a colon (:).
* The code blocks corresponding to if and else have to be indented (i.e. leading spaces at the begining of the line).



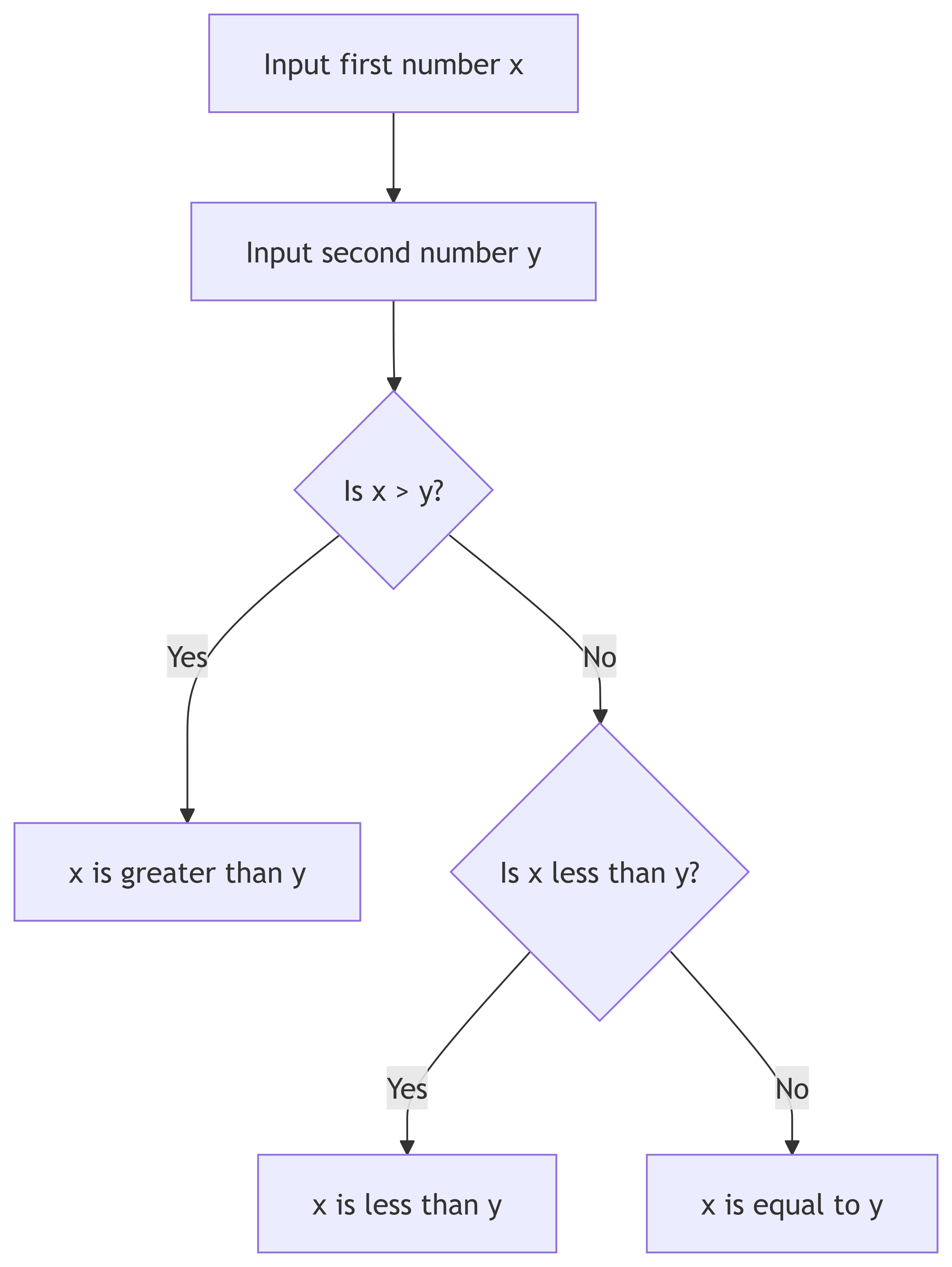
Flowchart to find greater of the two numbers.

x = 5  
y = 6  
  
if(x>y):  
 print("X is greater than Y")  
else:  
 print("X is less than Y")

X is less than Y

## 6.2 elif

Multiple conditions can be tested as well the elif keyword. The syntax for this keyword is similar to the if keyword i.e. elif also takes a condition within parenthesis. The else keyword take no argument such that its code block gets executed when the if condition (and elif) conditions are return False.

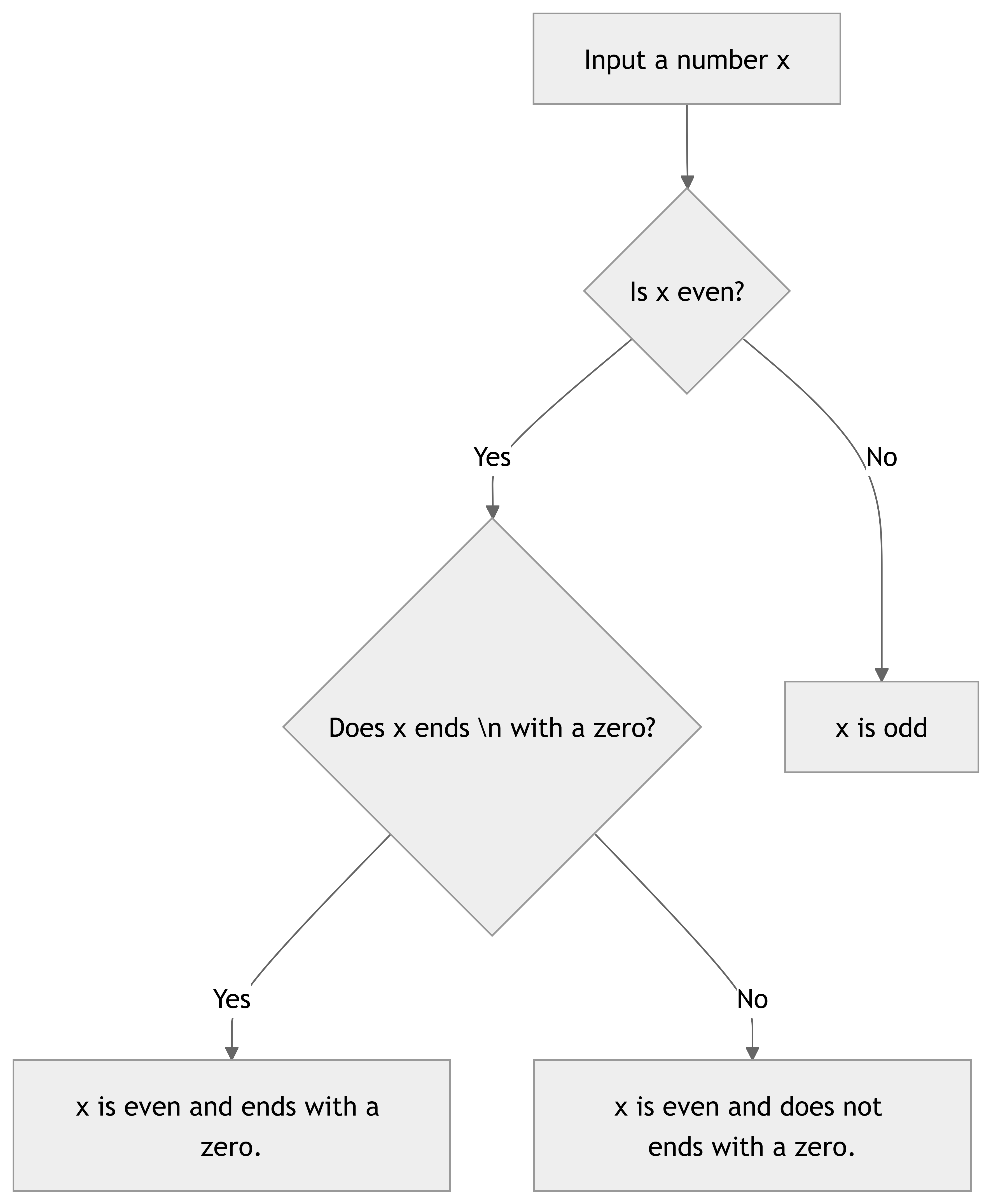


Flowchart to compare two numbers.

if(x>y):  
 print("X is greater than Y")  
elif(x<y):  
 print("X is less than Y")  
else:  
 print("X is equal to Y")

## 6.3 Nested conditional statements

For situations when there is a need to check multiple conditions in tandem, we can have nested conditional statements. This is different from the elif statement since here we need to check certain condition *after* some condition has already been tested to be true (or false depending upon the code). For instance, let’s say, we want to write a program that checks whether a number is even or odd and, if even, check whether the number ends with a zero.



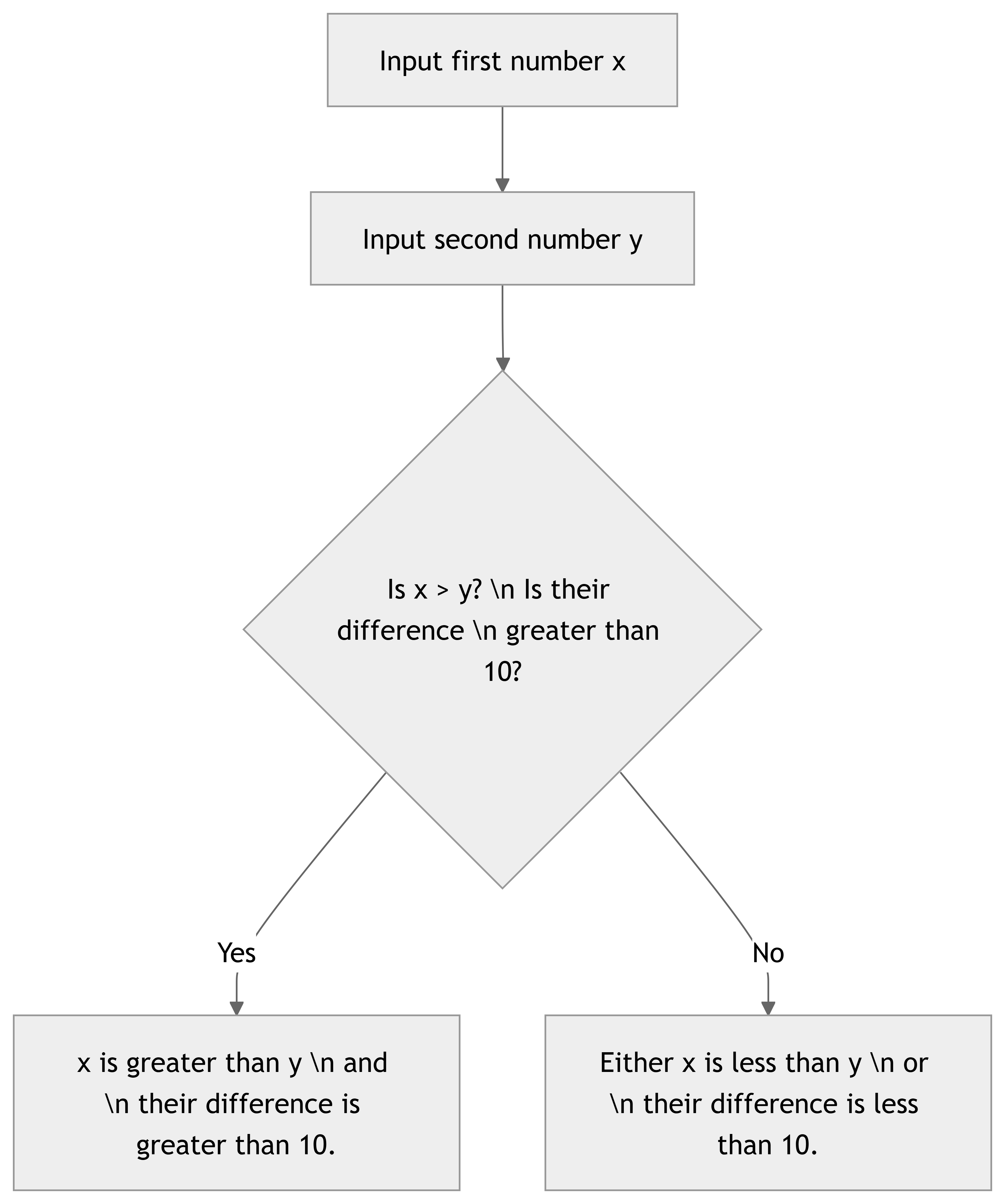
Flowchart for nested conditions.

x = 50  
if(x % 2 == 0):  
 if (x % 10 == 0):  
 print("The number is even and ends with a zero.")  
 else:  
 print("The number is even and does not ends with a zero.")  
else:  
 print("The number is odd.")

The number is even and ends with a zero.

## 6.4 Conditionals with Boolean operators

There are situations when multiple conditions need to be tested simultaneously. For such cases, we can use the Boolean operators – and, or, and not – with the if statement. To understand this let’s write a code to test, given two numbers, which number is greater and is the difference between the two numbers greater than 10.



Flowchart for comparison with Boolean logic.

x = 20  
y = 5  
  
if((x > y) and (x-y > 10)):  
 print("x is greater than y and the difference in x and y is more than 10.")  
else:  
 print("Either x is less than y or the difference in x and y is less than 10 or both.")

x is greater than y and the difference in x and y is more than 10.

Quiz: Write a program to check if a number is completely divisible by five.

x = 15  
if(x % 5 == 0):  
 print("The number is completely divisible by five")  
else:  
 print("The number is not completely divisible by five")

Quiz: Given a date of birth, write a program to check in which quater of the year the person was born.

# Note that the date of birth should be in a specific format e.g. DD/MM/YYYY.  
DOB = "14/02/2000"  
month = int(DOB.split("/")[1]) #change to integer data type.  
if(month < 4):  
 print("The person was born in the first quater")  
elif(month > 3 and month < 7):  
 print("The person was born in the second quater")  
elif(month > 6 and month < 10):  
 print("The person was born in the third quater")  
else:  
 print("The person was born in the fourth quater")

## 6.5 Using in with conditionals

So far we have seen the use of conditional operators for framing the if statements. When working with lists (or for that matter strings) we can use the in keyword to check for the presence of a particular element (or character).

fruits = ["Apple", "Mango", "Banana"]  
if("Mango" in fruits):  
 print("Make a mango shake")  
else:  
 print("No mango shake")

Make a mango shake

x = "abcde"  
if("c" in x):  
 print("c is there")  
else:  
 print("c is not there")

c is there

## 6.6 Functions that return a Boolean value

We can also construct the conditional statements using functions that return a Boolean value.

x = "Apple"  
if(x.startswith("A")):  
 print("The first alphabet of the variable x is A")  
else:  
 print("The first alphabet of the variable x is not A")

The first alphabet of the variable x is A

# The isinstance function checks if an object is an instance of a given class.  
  
y = 5.0  
if(isinstance(y, int)):   
 print("y is an integer")  
else:  
 print("y is not an integer")

y is not an integer

# 7. Loops

In programming often times there is a requirement that a particular set of instructions is required to be repeated certain number of times. This can be easily achieved using loops. These statements execute a code block repeatedly for specified number of times or until a condition is met.

## 7.1 For loop

The most common loop that is used to repeat a block of code for fixed number of times is for loop. The example below show iterating through all the elements of a [list](lists.html). Syntactically, the for code block is similar to the if block in terms of the use of colon and indentation.

list\_of\_numbers = [1, 2, 3, 4, 5]   
for x in list\_of\_numbers:  
 print(x\*2)  
print("Done")

2  
4  
6  
8  
10  
Done

The for statement begins with the for keyword which is followed by a variable name (x in the example above). This variable would store the different elements in the list as the loop progressess. The in keyword connects the variable with the list and finally we have the list that we want to iterate. The colon (:) at the end is required. It indicates the begining of the block of statements that would be part of the for loop. Next, we have a set of statement that needs to executed for each element of the list. The statement that are part of the for loop must begin with indentation (tab or some spaces). In the example above, the print(x\*2) statement is part of the for loop (since it is indented) while the print("Done") statement is not part of the for loop since it is not indented. There must be atleast one statement which is part of the for loop (i.e. atleast one statement that starts with indentation).

## 7.2 Keeping track of indices during iteration

In the code above, we were able to iterate through the items in a list using a for loop and perform some action on each of those items. However, there was no option to keep track of the index of an item during iteration. For cases where we need a list of items along with their indices, we have a couple of ways to achieve that. One way is to iterate through the range of numbers till the length of the list such the variable in the for loop will have the index of the items and the items themselves can be accessed by indexing the list, as shown below.

nums = [10,20,30,40,50]  
for x in range(len(nums)):  
 print(x, nums[x]) # use x+1 to start from 1 instead.

0 10  
1 20  
2 30  
3 40  
4 50

Another way is to use the enumerate function with a list as an argument. This function returns an enumerate object having a collection of tuples – each of which has the index and the corresponding item in the list. When using this in a for loop, we need to provide two variables to store the two values from the tuple.

nums = [10,20,30,40,50]  
for x,y in enumerate(nums):  
 print(x,y)

0 10  
1 20  
2 30  
3 40  
4 50

The default indexing for the enumerate function start with zero; which can be changed using the start keyword argument.

nums = range(10,51,10)  
for x,y in enumerate(nums, start=1):  
 print(x,y)

1 10  
2 20  
3 30  
4 40  
5 50

## 7.3 Iterating through multiple lists simultaneously

The zip function comes handy when the objective is to iterate through multiple lists (or other iterable python objects) at the same time. It returns a collection of tuples such that the nth tuple in this collection has nth element from all the lists passed as an argument. If the lists (which are passed as an argument to zip) are of unequal length then the iteration continues till the length of the shortest list. When running a for loop with zip, the number of variable should be equal to the number of lists passed to zip such that each items gets assigned to a different variable. A single variable can also be used in the for loop, in that case, that variable would be a tuple having the corresponding items.

fruits = ["Apple", "Banana", "Banana"]  
desserts = ["pie", "pie", "shake"]  
for x,y in zip(fruits, desserts):  
 print(x,y)

Apple pie  
Banana pie  
Banana shake

for x in zip(range(10,51,10),"abcde"):  
 print(x[0],x[1])

10 a  
20 b  
30 c  
40 d  
50 e

Quiz: Write a program that takes a list of 10 numbers as input and prints the cube of each of the number in the list.

x = range(1,11)  
for y in x:  
 print(y\*\*3)

We can also check multiple conditions at the same time. For this the Boolean operators (and, or, and not) are used along with the different conditions.

## 7.4 List comprehension

When we need to perform one or two operations on all the elements of a list, we have an option to write the for loop in one line. This is referred as list comprehension.

nums = range(1,6)  
for x in nums:  
 print(x\*\*2)  
  
# Using list comprehension  
nums\_squared = [x\*\*2 for x in nums]  
print(nums\_squared)

1  
4  
9  
16  
25  
[1, 4, 9, 16, 25]

We can also have conditional statements when using list comprehension.

nums = range(1,6)  
num\_even\_square = [x\*\*2 for x in nums if x%2==0]  
num\_even\_square\_odd\_cube = [x\*\*2 if x%2==0 else x\*\*3 for x in nums]  
print(num\_even\_square)  
print(num\_even\_square\_odd\_cube)

[4, 16]  
[1, 4, 27, 16, 125]

## 7.5 While loop

While loop is another frequently used construct to perform task repeatedly when the number of iterations is not fixed. This loop is executed till certain condition is met irrespective of number of iterations.

While expression:

    code block

While loop implicitly has a if conditional statement.

x = 5  
while(x>2):  
 print(x)  
 x = x-1  
print("Loop is over")

5  
4  
3  
Loop is over

When writting a while loop you must ensure that the condition is met at some point within the loop otherwise the loop will iterate infinitely. E.g. in the above code if the condition within while loop is changed to x>2 then it result in an infinite loop.

## 7.6 Controlling the execution of loops

Many a times it is require to skip the execution of loop for certain steps or to terminate the loop altogether. Python has some reserved keyword to facilitate this task. continue can be used to jump to subsequent iteration of the loop and break can be used to end the loop. These special keywords are generally used in conjuction with conditional statements to manipulate the execution of the loops.

for x in range (1,10):  
 if (x == 5):  
 continue  
 print(x)  
print("Done")

1  
2  
3  
4  
6  
7  
8  
9  
Done

for x in range (1,10):  
 if (x == 5):  
 break  
 print(x)  
print("Done")

1  
2  
3  
4  
Done

Quiz: Take a list of first 10 numbers and print the square for even numbers.

for x in range(1,11):  
 if(x%2 == 0):  
 print(x\*\*2)  
 else:  
 continue

# 8. Tuples and Sets

Python has different data types to store a collection of items. Tuples and sets are two such data types. Both these data have some unique characteristics that make them useful under different contexts.

## 8.1 Tuples - the immutable collection

A tuple is a collection of items just like a list; with one important difference that this collection is immutable. In other words, there is no option to add or remove elements form a tuple. A tuple can have elements of same data type or multiple data types. A tuple is denoted by parantheses (()). A tuple can be initialized by putting a comma-separated collection of element within parantheses (or even with paranthesis). The tuple keyword can be used to convert a list to a tuple. Note that when initializing a tuple with single element we need to put a comma after the element. To access an element by its index we can specify index in square brackets after the tuple name. A slice of a tuple returns a tuple.

fruits = ("apple", "banana", "mango")  
print(fruits)  
print(type(fruits))  
print(fruits[1])

('apple', 'banana', 'mango')  
<class 'tuple'>  
banana

fruits2 = "pineapple", "cherry"  
print(fruits2)

('pineapple', 'cherry')

nums = [1,2,3,4,5]  
nums\_tuple = tuple(nums)  
print(nums\_tuple)

(1, 2, 3, 4, 5)

Tuple can be concatenated in the same manner as strings are concatenated. The + and \* operators can be used for tuple concatenation.

print(fruits+fruits2)  
print(fruits2\*3)

('apple', 'banana', 'mango', 'pineapple', 'cherry')  
('pineapple', 'cherry', 'pineapple', 'cherry', 'pineapple', 'cherry')

Quiz: What would be the output of the following code. fruits = (“apple”, “mango”) fruits.append(“grapes”)      
 print(fruits)

Error. Tuples are immutable.

### 8.1.1 Advantages of Tuples

The fact that tuples are immutable is an important utility feature for this collection data type. For instance, the keys of a dictionary must be of immutable data type, so we can use tuples to have a collection of items as a key.

# a dictionary with keys having studnets name and roll number and   
# values having the subjects taken.  
students\_data = {('Sam', 102): ['Physics', 'Maths', 'Chemistry'],  
 ('Mohan', 103): ['Physics', 'Chemistry', 'Biology']}  
for k1, v1 in students\_data.items():  
 print(f'{k1[0]} (Roll No.{k1[1]}) has taken {", ".join(v1)}.')

Sam (Roll No.102) has taken Physics, Maths, Chemistry.  
Mohan (Roll No.103) has taken Physics, Chemistry, Biology.

An important point to note here is that when the tuples are used as a dictionary key then all the elements of a tuple should be immutable. If a tuple with an mutable element (e.g. a list) is used as a dictionary key then it would lead to a TypeError.

## 8.2 Sets and their characteristics

A set is an another data type, just like lists and tuples, that can be used to store a collection of items. A set object can be initialized using the set function or using curly braces {} with elements inside. Note that using empty curly braces would initialize a dict object. The set function take a list as an argument for initializing a set. A notable difference between sets and other collections is that in sets the items are unordered. This implies that we cannot access individual elements of a set by their indices.

Following are characteristics of sets: - Elements in a set are unordered and duplicate elements are not allowed. - Elements in a set must be of immutable data types only. - Elements can be added to and removed from a set.

a = {"apple", "banana", "mango"}  
print(type(a))  
for element in a:  
 print(element)

<class 'set'>  
banana  
mango  
apple

The add and update functions can be used to add one or multiple elements to a set, respectively. To remove an element from a set, discard or remove functions can be used with element as an argument. The former would leave the set unchanged if an element is not present in that set, while the later would lead to an error.

a.add("cherry")  
print(a)  
a.remove("cherry")  
print(a)

{'banana', 'mango', 'apple', 'cherry'}  
{'banana', 'mango', 'apple'}

### 8.2.1 Set operations

We can apply different logical operations to parse content of multiple sets. For instance, to get all the elements present in one set and not present in another set, we can use the minus - operator. Similarly, we can get the union and intersection of two sets using the | and & operators respectively. The carrot (^) operator return the unique elements in the two sets.

a = {"apple", "banana", "mango"}  
b = {"apple", "banana", "cherry"}  
print(a,b)  
print("Elements present in a and not in b", a-b)  
print("Elements present in b and not in a", b-a)  
print("Union of sets a and b", a|b)  
print("Intersection of sets a and b", a&b)  
print("Unique elements in sets a and b", a^b)

{'banana', 'mango', 'apple'} {'banana', 'apple', 'cherry'}  
Elements present in a and not in b {'mango'}  
Elements present in b and not in a {'cherry'}  
Union of sets a and b {'banana', 'mango', 'apple', 'cherry'}  
Intersection of sets a and b {'banana', 'apple'}  
Unique elements in sets a and b {'mango', 'cherry'}

Quiz: Which of the following expression is equivalent to a^b. a-b&a+b a-b|b-a a|b-a&b a&b-a|b

a-b|b-a

# 9. Dictionaries

These are collection of items arranged as a key-value pair. A dictionary is represented by {} and there is a dict keyword in python to assign dictionary data type. The value for a particular key in a dictionary can be accessed by specifing the key within [] after the dictionary name.

colors = {'blue':'sky', 'white':'milk', 'red':'rose'}  
print(colors)  
print(type(colors))  
print(colors["red"])

{'blue': 'sky', 'white': 'milk', 'red': 'rose'}  
<class 'dict'>  
rose

An alternate way to initialize a dictionary is by using the dict keyword. In the code below, notice the differences in how the key-value pairs are specified while inititalize a dictionary with the dict keyword. The len function with a dictionary as an argument returns the total number of key-value pairs in that dictionary.

colors\_new = dict(blue="ink", white="paper")  
print(colors\_new)  
print(len(colors\_new))

{'blue': 'ink', 'white': 'paper'}  
2

The values in a dictionary could any data type such as string, integer, list or even dictionary. On the other hand, keys can only be immutable data types such as string, integer or tuple.

# a dictionary with numbers as keys  
Numbers = {1:"one", 2:"two"}  
print(Numbers[1])

one

# a dictionary with list as values  
stationery = {"pens":["ball", "fountain"], "paper":["A4", "A3", "A2"]}  
print(stationery["paper"])  
print(stationery["paper"][1])

['A4', 'A3', 'A2']  
A3

## 9.1 Iterating through dictionaries

The keys and values functions can be used to get all keys and all values as a list, respectively. To get all the keys of a dictionary as list, the list function with dictionary as an argument can also be used.

print(stationery.keys())  
print(stationery.values())  
print(list(stationery))

dict\_keys(['pens', 'paper'])  
dict\_values([['ball', 'fountain'], ['A4', 'A3', 'A2']])  
['pens', 'paper']

The items function comes handy for iterating through the content of the dictionary. This function return the key-value pairs as a tuple and all these key-value tuples are arranged in a list. We can then iterate through this list to access the key-value pairs. Note that while iterating through this list, two variables need to be specified in the for loop which would be mapped to the key and value in the tuple, respectively.

print(stationery.items())

dict\_items([('pens', ['ball', 'fountain']), ('paper', ['A4', 'A3', 'A2'])])

for k1, v1 in stationery.items():  
 print(f"The key is {k1} and its value is {v1}")

The key is pens and its value is ['ball', 'fountain']  
The key is paper and its value is ['A4', 'A3', 'A2']

Just like lists, we can modify the contents of a dictionary by adding or removing key-value pairs. When adding a key-value pair if the key doesn’t exists then a new key would be created while if the key is already present then its value would be modified.

Numbers = {1:"one", 2:"two"}  
print(Numbers)  
Numbers[3] = "three"  
print(Numbers)  
Numbers[2] = "dos"  
print(Numbers)

{1: 'one', 2: 'two'}  
{1: 'one', 2: 'two', 3: 'three'}  
{1: 'one', 2: 'dos', 3: 'three'}

del(Numbers[2])  
print(Numbers)

{1: 'one', 3: 'three'}

## 9.2 Dict comprehensions

Yet another way to initialize a dictionary is via dict comprehensions. Just like list comprehensions, here a dictionary is created by processing elements in a list. The code below creates a dictionary with numbers and their squares as key-value pairs.

new\_dict = {x: x\*\*2 for x in range(1,5)}  
print(new\_dict)

{1: 1, 2: 4, 3: 9, 4: 16}

Quiz: Given a dictionary: d1 = {1:“one”, 2:“two”, 3:“three”} Write a code to check if “two” is present 1) as a key and 2) as a value.

d1 = {1:"one", 2:"two", 3:"three"}  
if("two" in d1.keys()):  
 print('"two" is a key in d1')  
if("two" in d1.values()):  
 print('"two" is a value in d1')  
   
# Alternate solution  
print("Alternate solution")  
for k1,v1 in d1.items():  
 if("two" == k1):  
 print('"two" is a key in d1')  
 if("two" == v1):  
 print('"two" is a value in d1')

"two" is a value in d1  
Alternate solution  
"two" is a value in d1

# 10. File Handling

Saving and retrieving information is frequently required when working with computer programs. Python has methods to read and write files. In addition, there are some libraries that can facilitate working with specialized file types. For reading a file, a file handler needs to be initiated which is an object that act as an interface to that file. A file handler incorporates information about the file along with its path and the mode in which the file is available for processing. The mode here refers to the way the file would be opened i.e. for reading, writing, etc. The open function takes name of the file as an argument along with mode and returns the corresponding file handler. This file handler object is iterator and can be considered as a collection having of all the lines in the file. Once done with reading the contents for the file, the file handler must be closed using the close function.

## 10.1 Reading files

To read a file using the open function, we will pass the file name as the first argument and the mode i.e. r are the second argument. Note that when we given only the file name as an argument, the file should be present in the current working directory. If a file from another folder needs to be accessed then we need to give the full path for that file. To check the working directory path, pwd command can be used.

pwd

'C:\\Users\\bioinfo guru\\OneDrive - bioinfo.guru\\Documents\\python\_book'

We can used the Jupyter magic command %load to display the contents a text file within the notebook as shown below.

# %load test\_file.txt  
This is a test file for Python.  
This file has .txt extension.

# test\_file.txt should be present in the current working directory.  
  
FH1 = open('test\_file.txt','r')  
  
print(f"The file {FH1.name} is open in {FH1.mode} mode.")  
  
# print all the lines in the file.  
for x in FH1:  
 print(x)  
FH1.close()

The file test\_file.txt is open in r mode.  
This is a test file for Python.  
  
This file has .txt extension.

Often it is useful to remove the newline character (\n) at the end of the lines since the print fuction add as newline by default. This can be achieved using the rstrip() function. This function without any argument remove the last character in the string and return a copy of the modified string.

## 10.2 Writing files

For writing content to a file the open function should be called with ‘w’ as the mode. When a file is opened in the write mode, a new file is created. In case there the file already exists then its contents are overwritten (without warning!). write function is used to write content to a file. We can also use print with file attribute to write to file instead of printing on screen.

FH\_out = open('temp\_file.txt', 'w')  
FH\_out.write("This is the first sentence.\n")  
print("This is the second sentence.", file=FH\_out)  
FH\_out.close()

# %load temp\_file.txt  
This is the first sentence.This is the second sentence.

FH2 = open('temp\_file.txt','r')  
for lines in FH2:  
 print(lines)  
FH2.close()

This is the first sentence.  
  
This is the second sentence.

To append contents to an existing file, it should be opened with ‘a’ option instead of ‘w’.

FH\_out = open('temp\_file.txt','a')  
FH\_out.write("This is the third sentence.\n")  
FH\_out.write("This is the \t fourth sentence.\n")  
FH\_out.close()

FH3 = open('temp\_file.txt','r')  
for lines in FH3:  
 lines = lines.rstrip('\n')  
 print(lines)  
FH3.close()

This is the first sentence.  
This is the second sentence.  
This is the third sentence.  
This is the fourth sentence.

The readlines function can be used to get a list having lines of the file as elements.

FH4 = open('temp\_file.txt','r')  
all\_lines = FH4.readlines()  
print(all\_lines)  
FH4.close()

['This is the first sentence.\n', 'This is the second sentence.\n', 'This is the third sentence.\n', 'This is the \t fourth sentence.\n']

## 10.3 The with keyword

We can also read and write file using the with keyword. Here some action are performed on the file object within the with block. This approach automatically closes the when the with block is over.

with open("temp123.txt", "w") as FH\_OUT:  
 print("Hello", file=FH\_OUT, end=" ")   
 print("World!", file=FH\_OUT)  
print("Done")

Done

with open("temp123.txt", "a") as FH\_OUT:  
 print("hi", file=FH\_OUT)

with open("temp123.txt","r") as FH:  
 for line in FH.readlines():  
 print(line)

Hello World!  
  
hi

Quiz: Write a program to print the third line of a text file.

#temp\_file.txt should be there in the current directory  
FH = open('temp\_file.txt','r')  
all\_lines = FH.readlines()  
print(all\_lines[2])

## 10.4 Working with multiple files

Many a times we need to process multiple files such that we cannot use individual file names to open and therefore we need a way to programmatically access all the files in a folder. To do this, we have a few options in Python. Libraries such as os and glob provides functionality to iterate over files in a folder. Using the glob function from the glob library, the code below creates a list of files with name starting with “temp” and having a “.txt” extension.

import glob  
txt\_files = glob.glob("temp\*.txt")  
print(txt\_files)

['temp123.txt', 'temp\_file.txt']

We can now iterate through this list of files and print their contents. The line\_count variable stores the number of lines in each file and is printed after iterating through the contents of a particular file.

for f in txt\_files:  
 line\_count = 0  
 print(f'Reading file {f}')  
 FH = open(f, 'r')  
 for line in FH:  
 print("\t"+line.rstrip("\n"))  
 line\_count += 1  
 FH.close()  
 print(f'{f} has {line\_count} line(s).')

Reading file temp123.txt  
 Hello World!  
temp123.txt has 1 line(s).  
Reading file temp\_file.txt  
 This is the first sentence.  
 This is the second sentence.  
 This is the third sentence.  
 This is the fourth sentence.  
temp\_file.txt has 4 line(s).

# 11. NumPy

A Python library for scientific computing. It features multidimensional array objects along with an assortment of functions optimized for fast computation. To use this library we first need to import it as shown below. As a convention, the numpy library is imported with an alias np. This will give us access to all the functions within the NumPy library. Let’s create a numpy array.

import numpy as np  
a1 = np.array([1,2,3,4,5,6])  
print(a1)  
print(type(a1))

[1 2 3 4 5 6]  
<class 'numpy.ndarray'>

Although the array looks like a list but it has very different properties and functions associated with it as compared to the python list. An important difference between a list and an ndarray is that the former can have elements of muliple data types while the later can only contain elements of same data type. There are multiple built-in function to create and manipulate arrays for numerical analysis. Let’s have a better understanding of the differences between lists and ndarrays by looking at some the functions associated with arrays. For instance, there are no in-built functions to change the shape of a list or to transpose a list whereas these operations can be done seemlessly with ndarrays. The shape function returns the dimensions of an ndarray as a tuple and the reshape function changes the shape of an ndarray given a tuple.

print("The shape of a1 array is", a1.shape)  
m1 = a1.reshape(2,3)  
print(m1)  
print("The shape of m1 array is", m1.shape)

The shape of a1 array is (6,)  
[[1 2 3]  
 [4 5 6]]  
The shape of m1 array is (2, 3)

At times, there is a need to change the orientation of the data i.e. the rows in a matrix should be arranged as columns. This can be achieve using the transpose functions. The .T attribute of a numpy array does the same operation. In addition, to get all the values of a multidimensional as a 1D array, use the ravel function. This flattening of multidimensional array is useful when we want to iterate over all the elements in an array without considering the dimensions. The output of ravel function is equivalent to the output of reshape(-1).

print(m1)  
m1\_transposed = m1.T  
print(m1\_transposed)

[[1 2 3]  
 [4 5 6]]  
[[1 4]  
 [2 5]  
 [3 6]]

print(m1)  
m1\_flattened = m1.ravel()  
print(m1\_flattened)

[[1 2 3]  
 [4 5 6]]  
[1 2 3 4 5 6]

There are various functions available in NumPy to create arrays e.g. zeros, ones, empty, random.rand can be used to create an ndarray having all the values as zero, one, no value, or random values, respectively. These three functions take shape of the array as an argument and data type can be optionally specified. arange is another function that can be used to initialize an array with specific values. This function is similar to the range function with a difference that it return an array while the range function returns a list.

|  |
| --- |
| [[0. 0. 0.]  [0. 0. 0.]  [0. 0. 0.]] [[0.17241597 0.28930394 0.79425668]  [0.18228287 0.33953533 0.03627949]  [0.75375968 0.18508584 0.30824166]] |

array1 = np.arange(0,10)  
print(array1)  
array2 = np.arange(0,10,2)  
print(array2)

[0 1 2 3 4 5 6 7 8 9]  
[0 2 4 6 8]

When using the arange function, we need to specify the spacing (step) between the numbers that we want to generate within a range of numbers. Let’s say, we need to generate a *specific number* of equally-spaced points within a range of two numbers. Now arange function is not suitable since there is no option to specify total number of points. In this case, linspace function can be used which takes three numbers as argument, the first two numbers specify the range (both number included) and the third number specifies the number of values to return. The linspace function automatically figures out the ‘step’ or spacing between the numbers.

array3 = np.linspace(0,3,5)   
print(array3)  
array4 = np.linspace(1,10,4)  
print(array4)

[0. 0.75 1.5 2.25 3. ]  
[ 1. 4. 7. 10.]

## 11.1 Reading data from file

Numpy has genfromtxt function that comes handy for reading data from text files. The example below show read a csv file using this function. The file test1.csv has five rows with each row having three values separated by comma.

# %load test1.csv  
1, 1, 1  
2, 4, 8  
3, 9, 27  
4, 16, 64  
5, 25, 125

inp\_data = np.genfromtxt("test1.csv", delimiter=",")  
print(inp\_data)  
print(inp\_data.shape)

[[ 1. 1. 1.]  
 [ 2. 4. 8.]  
 [ 3. 9. 27.]  
 [ 4. 16. 64.]  
 [ 5. 25. 125.]]  
(5, 3)

# slice of the matrix  
sub\_matrix = inp\_data[1:4]  
print(sub\_matrix)

[[ 2. 4. 8.]  
 [ 3. 9. 27.]  
 [ 4. 16. 64.]]

## 11.2 Writing data to file

Numpy has savetxt function that can be used to save numpy arrays to a text file in e.g. csv format. There is an option to change the delimiter. The example below shows writing a csv file using this function with space as a delimiter. The fmt argument is used to specify the format in which to write the data.

np.savetxt("test2.csv",sub\_matrix, delimiter=" ", fmt='%d')

# %load test2.csv  
2 4 8  
3 9 27  
4 16 64

## 11.3 Slicing multidimensional arrays

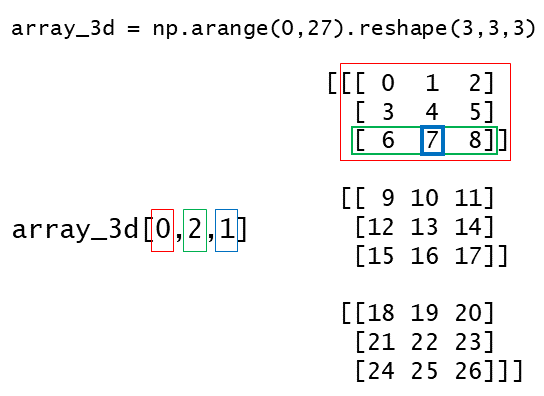
Just like we can splice a string or list, slicing of numpy arrays can also be performed. While in case of a sting or a list the slice operator takes the start and end positions, in the case of ndarrays, the slice operator would take as many start and end combinations as the dimensions of the ndarray. Which means that slicing can be performed for each dimension of a multidimensional ndarray. In the example below, we first create a three dimensional array having values from 0 to 27. Note that the zeroth element of this 3D array is a 2D array (with values 0 to 8). Similarly, the first element of this 3D array is another 2D array (with values 9 to 17). Next, using slicing, we’ll print one of the number of from this 3D array.

array\_3d = np.arange(0,27).reshape(3,3,3)  
print(array\_3d)

[[[ 0 1 2]  
 [ 3 4 5]  
 [ 6 7 8]]  
  
 [[ 9 10 11]  
 [12 13 14]  
 [15 16 17]]  
  
 [[18 19 20]  
 [21 22 23]  
 [24 25 26]]]

print(array\_3d.shape)  
print(array\_3d[0,2,1])

(3, 3, 3)  
7



The figure shows the parsing of the slice operator for a 3D array.

print(array\_3d[:,:,0])

[[ 0 3 6]  
 [ 9 12 15]  
 [18 21 24]]

Quiz: What would be the slice operator for array\_3d to get the following output [[[10 11]   [13 14]]  [[19 20]   [22 23]]]

print(array\_3d[1:3,:2,1:3])

## 11.4 Combining arrays

The Numpy arrays can be combined using concatenate function. All the arrays that are to be combined must have same number of dimensions. The arrays to be joined as passed as a tuple and an axis can be specified to indicate the direction along which to join the arrays. The arrays can be joined along row-wise or column-wise by specifing axis as 0 or 1, respectively. Note that the dimensions for all the arrays along the concatenation axis must be identical. That is to say that, e.g., when combining two or more arrays along rows - the number of rows in all the arrays should be same.

print(np.arange(1,5))

[1 2 3 4]

arr1 = np.arange(1,5).reshape(2,2)  
arr2 = np.array([[5,6]]) #note the square brackets  
  
print(arr1.shape)  
print(arr2.shape)

(2, 2)  
(1, 2)

arr3 = np.concatenate((arr1,arr2), axis=0)   
print(arr3)

[[1 2]  
 [3 4]  
 [5 6]]

The above code won’t work with axis=1 because the number of rows in the two arrays is not same. We can transpose the arr2 and then concatenate it with arr1 along axis=1.

arr4 = np.concatenate((arr1, arr2.T), axis=1) #note the .T  
print(arr4)

[[1 2 5]  
 [3 4 6]]

Similar, output can be generated using the vstack and hstack functions as shown below.

print(np.vstack((arr1,arr2)))  
print(np.hstack((arr1,arr2.T)))

[[1 2]  
 [3 4]  
 [5 6]]  
[[1 2 5]  
 [3 4 6]]

## 11.5 Broadcasting

When performing arithmetic operations with arrays of different sizes, numpy has a an efficient way of broadcasting the array with lower dimensions to match the dimensions of larger array. Broadcasting can be thought of as creating replicas of the original array. This vectorized array operation is a lot faster than conventional looping in Python.

arr1 = np.arange(1,5).reshape(2,2)  
arr2 = np.array([[5,6]]) #note the square brackets  
print(arr1)  
print(arr2)

[[1 2]  
 [3 4]]  
[[5 6]]

When we multiply the two arrays (arr1 \* arr2) the arr2 would be broadcasted such as its shape would change from (1,2) to (2,2). Now, with this broadcasted array the multiplication would be performed element-wise. Similarly, when with multiple a scaler with a ndarray, the scaler is broadcasted to match the dimensions of the ndarray followed by element-wise multiplication.

print(arr1\*arr2) #arr2 would be broadcasted along rows

[[ 5 12]  
 [15 24]]

print(arr1\*arr2.T) #arr2 would be broadcasted along columns

[[ 5 10]  
 [18 24]]

print(arr1\*2)

[[2 4]  
 [6 8]]

Quiz: Write a program to calculate cube of first 10 natural numbers.

print(np.arange(1,11)\*\*3)

This ability to broadcast open up lot of possibilities when working with martices. A simple example is shown be to print table for first ten natural numbers.

num1 = np.arange(1,11).reshape(1,10)  
all\_ones = np.ones((10,10), dtype=int)  
table\_10 = all\_ones\*num1\*num1.T  
print(table\_10)

[[ 1 2 3 4 5 6 7 8 9 10]  
 [ 2 4 6 8 10 12 14 16 18 20]  
 [ 3 6 9 12 15 18 21 24 27 30]  
 [ 4 8 12 16 20 24 28 32 36 40]  
 [ 5 10 15 20 25 30 35 40 45 50]  
 [ 6 12 18 24 30 36 42 48 54 60]  
 [ 7 14 21 28 35 42 49 56 63 70]  
 [ 8 16 24 32 40 48 56 64 72 80]  
 [ 9 18 27 36 45 54 63 72 81 90]  
 [ 10 20 30 40 50 60 70 80 90 100]]

num1

array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]])

#Print tables as seen in math books  
table\_arr = np.empty((5,1), dtype="<U200")  
num2 = np.arange(1,6)  
for x in num2:  
 for y in num2:  
 s = f"{x:>2} X {y:>2} = {x\*y:<4}"  
 table\_arr[y-1][0] = table\_arr[y-1][0]+s  
# iterate over the array to print each row as a string  
# the slice operator is used to remove square brackets and quotes  
for i in range(len(table\_arr)):  
 print(np.array2string(table\_arr[i])[3:-2])

1 X 1 = 1 2 X 1 = 2 3 X 1 = 3 4 X 1 = 4 5 X 1 = 5   
1 X 2 = 2 2 X 2 = 4 3 X 2 = 6 4 X 2 = 8 5 X 2 = 10   
1 X 3 = 3 2 X 3 = 6 3 X 3 = 9 4 X 3 = 12 5 X 3 = 15   
1 X 4 = 4 2 X 4 = 8 3 X 4 = 12 4 X 4 = 16 5 X 4 = 20   
1 X 5 = 5 2 X 5 = 10 3 X 5 = 15 4 X 5 = 20 5 X 5 = 25

## 11.6 Functions for ndarrays

Numpy comes with a lot of useful functions that facilitates working with ndarrays. Readers are encourages to refer to the [NumPy documentation](https://numpy.org/doc/stable/reference/routines.html) for details.

**where()**

The where function is used to get the indcies of values based on certain condition(s). The get the values corresponding to those indcies, we can slice the original array with the output indcies.

arr1 = np.linspace(0,100,10, dtype='int')  
print("arr1 indcies where the value is greater than 50:")  
print(np.where(arr1 > 50))  
print("arr1 elements where the value is greater than 50:")  
print(arr1[np.where(arr1 > 50)])

arr1 indcies where the value is greater than 50:  
(array([5, 6, 7, 8, 9], dtype=int64),)  
arr1 elements where the value is greater than 50:  
[ 55 66 77 88 100]

**sort()**

The values in a numpy array can be sorted along different axes using the sort function. Also, all the values can be sorted after flattening the array by passing the None to the axis argument. This function returns the sorted array and the original array remains unchanged.

arr2 = np.vstack((arr1,np.linspace(100,1,10, dtype='int')))  
print(arr2)

[[ 0 11 22 33 44 55 66 77 88 100]  
 [100 89 78 67 56 45 34 23 12 1]]

print("Sorting row-wise")  
print(np.sort(arr2, axis=1))  
print("Sorting column-wise")  
print(np.sort(arr2, axis=0))

Sorting row-wise  
[[ 0 11 22 33 44 55 66 77 88 100]  
 [ 1 12 23 34 45 56 67 78 89 100]]  
Sorting column-wise  
[[ 0 11 22 33 44 45 34 23 12 1]  
 [100 89 78 67 56 55 66 77 88 100]]

print("Sorting the flattened array")  
print(np.sort(arr2,axis=None))

Sorting the flattened array  
[ 0 1 11 12 22 23 33 34 44 45 55 56 66 67 77 78 88 89  
 100 100]

**Statistics**

Below are examples of some of the functions available in numpy for order and descriptive statistics. The axis argument can be used to calculate statistics along a specific dimension - zero means along the columns and one is for row-wise calculations.

# calculate range (max - min)  
# Here ptp means 'peak-to-peak'  
print(arr2)  
print("Column-wise range")  
print(np.ptp(arr2,axis=0))

[[ 0 11 22 33 44 55 66 77 88 100]  
 [100 89 78 67 56 45 34 23 12 1]]  
Column-wise range  
[100 78 56 34 12 10 32 54 76 99]

The percentile function return the q-th percentile for given data. This function take two positional arguments - data and q. The second argument (q-th percentile) can be an number or a list of numbers.

#calculate 50-th percentile  
print("50th percentile along the columns")  
print(np.percentile(arr2,50, axis=0))  
print("25th, 50th, and 75th percentile along the rows")  
print(np.percentile(arr2,[25,50,75], axis=1))

50th percentile along the columns  
[50. 50. 50. 50. 50. 50. 50. 50. 50. 50.5]  
25th, 50th, and 75th percentile along the rows  
[[24.75 25.75]  
 [49.5 50.5 ]  
 [74.25 75.25]]

#calculate mean, standard deviation, and variance  
print(np.mean(arr2, axis=0))  
print(np.std(arr2, axis=0))  
print(np.var(arr2, axis=0))

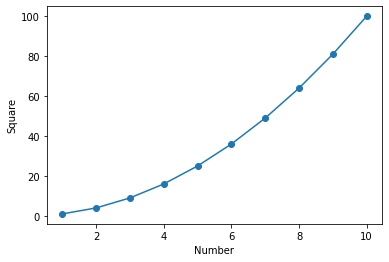
[50. 50. 50. 50. 50. 50. 50. 50. 50. 50.5]  
[50. 39. 28. 17. 6. 5. 16. 27. 38. 49.5]  
[2500. 1521. 784. 289. 36. 25. 256. 729. 1444.  
 2450.25]

# 12. Matplotlib

Matplotlib is a Python library for data visualization. It offers variety of functions to plot different types of graphs which can be customised to create high quality figures. The pyplot function in this library is used for instantiating a matplotlib graph object. The plot function draws a line plot given two sequences of equal length. The get the scatter plot use scatter. Properties such as line-width, line-color, marker-type, etc. can be easily customized using the appropriate keyword arguments. The pyplot object also has functions for decorating the plots with axis-labels, title, text, legend, etc. The show function renders the plot.

import matplotlib.pyplot as plt  
import numpy as np

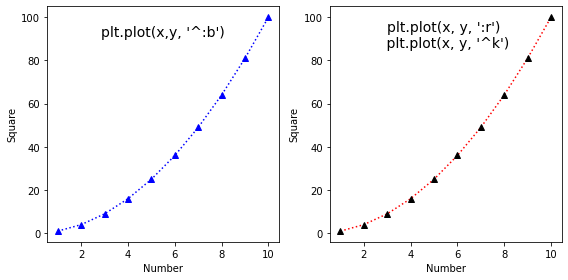
x = range(1,11)  
y = [a\*\*2 for a in x]  
plt.plot(x,y) #line plot  
plt.scatter(x,y) #scatter plot  
plt.xlabel("Number")  
plt.ylabel("Square")  
plt.show()



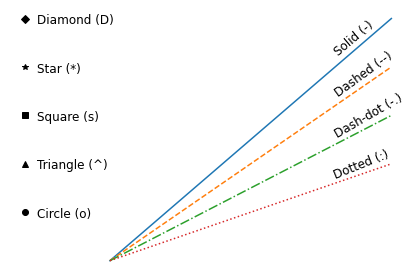
## 12.1 Styling plots

A plot rendered using the plot function can be styled using specific keyword arguments to control the layout and/or size of different elements within the plot. To customize the marker and line styles, marker and linestyle keyword arguments can be used. Similarly, color (or c) and linewidth (or lw) keyword arguments control the color and width of the line.

In addition, for the plot function, styling can also be done using a format string which specifies different styling attributes. The syntax for this format string comprises of a single letter code for marker style, line style, and color (preferably in this order). The “mlc” format string can be passed as an argument to the plot function. This format string can have all three components or less as requried. Note that if either marker style or line style character is missing then the plotting of marker or line is skipped. And if both marker and line styles are not there then a solid line is plotted with the specified color. The figure below shows data used above in two different representations. On left, the plot uses '^:b' format string while on the right graph uses two different format strings – ':r' (dotted line in red) and '^k' (triangle in black) – in the two calls to the plot function.

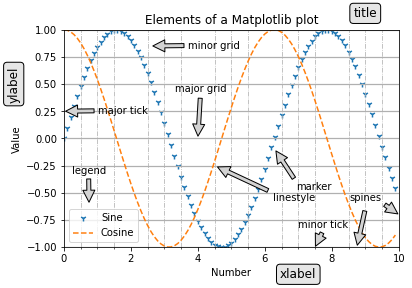


The image below shows some for the options for the marker and line styles available in matplotlib. The character given in parenthesis is used to plot the corresponding marker or line style. Note that these style can further be customized as required, for details please have look at the pyplot.plot [documentation](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html).



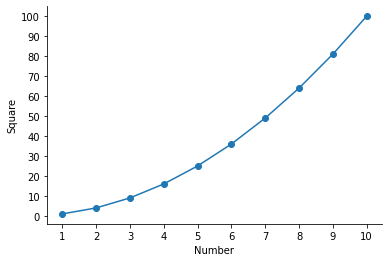
## 12.2 Components of a matplotlib figure

In addition to the styling options discussed above, matplotlib offers several option to control the aesthetics of a plot. The image below shows the different customizable elements in a matplotlib figure. These elements are accessible via different function available for the pyplot object or the axes object. The axes object is accessible through the gca function for the pyplot object.



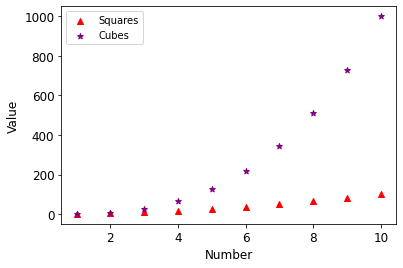
Some of the components of this figure can be modified directly using the pyplot object. E.g., the axes label can be changed using the xlabel and ylabel functions available for the pyplot object. However, to change the x-axis ticks, there is no direct way to do this using the pyplot object. So we need to create an axes object to modify x-ticks. The set\_ticks function available for the axes.xaxis object can ge used to specify a list having locations for xticks. Similarly, y-ticks can be customized using the corresponding function for the yaxis. We’ll also remove the top and right borders using spines atribute for the axes object.

x = range(1,11)  
y = [a\*\*2 for a in x]  
plt.plot(x,y) #line plot  
plt.scatter(x,y) #scatter plot  
plt.xlabel("Number")  
plt.ylabel("Square")  
ax = plt.gca()  
ax.xaxis.set\_ticks(range(1,11))  
ax.yaxis.set\_ticks(range(0,101,10))  
ax.spines[["right","top"]].set\_visible(False)  
plt.show()

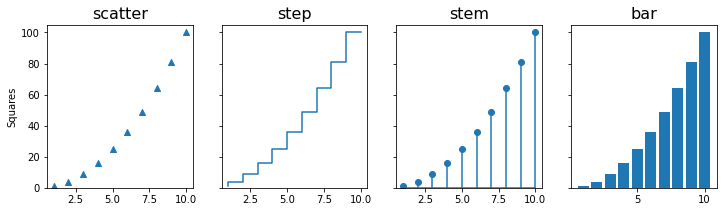


We can also have multiple plots in one graph i.e. different plots sharing the same axes.

x = range(1,11)  
y = [a\*\*2 for a in x]  
z = [a\*\*3 for a in x]  
plt.scatter(x,y,marker="^", color="red")  
plt.scatter(x,z,marker="\*", color="purple")  
plt.xlabel("Number", fontsize=12)  
plt.ylabel("Value", fontsize=12)  
plt.xticks(fontsize=12)  
plt.yticks(fontsize=12)  
plt.legend(["Squares","Cubes"])  
plt.show()



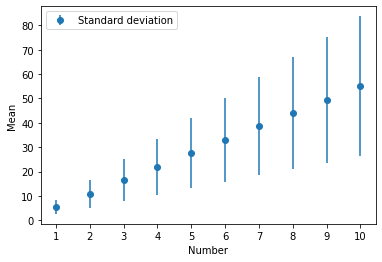
Matplotlib has a variety of in-built graph types that can be used to display information as required. Below are examples of some of the plots available in matplotlib. The title of these plots represent the corresponding function avaiable for the pyplot object.



## 12.3 Error bars

The errorbar function can be used to draw a plot with error bars. The yerr keyword argument specifies the size of the errorbars. By default, a line is drawn connecting the data points, which can be changed using the fmt argument. To plot error bars with ploting the data points, use fmt="none". In this example, we’ll use the 2D numpy array having table of numbers (see the chapter on [NumPy](Numpy.html) for details). The mean value along the rows of the ndarray are plotted as dots and standard deviation as the error bars.

num1 = np.arange(1,11).reshape(1,10)  
all\_ones = np.ones((10,10), dtype=int)  
table\_10 = all\_ones\*num1\*num1.T  
  
plt.errorbar(num1[0], np.mean(table\_10, axis=0), yerr=np.std(table\_10, axis=0),  
 label="Standard deviation", fmt="o")  
plt.legend(loc='upper left')  
plt.gca().xaxis.set\_ticks(num1.T)  
plt.xlabel("Number")  
plt.ylabel("Mean")  
plt.show()



We can also read a csv file which has the values for the data points and error bars in two columns. E.g. the code below reads a csv file and then make an errorbar plot using values in the two columns. The loadtxt function from numpy library is used to read the csv file in to a numpy array.

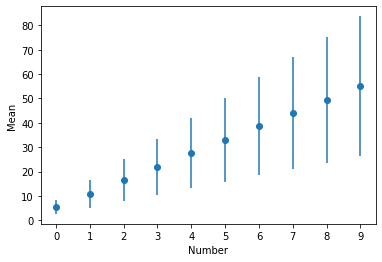
# %load mean\_std.csv  
Mean, Std  
 5.5, 2.87228132  
11.0, 5.74456265  
16.5, 8.61684397  
22.0, 11.48912529  
27.5, 14.36140662  
33.0, 17.23368794  
38.5, 20.10596926  
44.0, 22.97825059  
49.5, 25.85053191  
55.0, 28.72281323

data\_csv = np.loadtxt("mean\_std.csv", delimiter=',', skiprows=1) # skip the header row  
data\_csv

array([[ 5.5 , 2.87228132],  
 [11. , 5.74456265],  
 [16.5 , 8.61684397],  
 [22. , 11.48912529],  
 [27.5 , 14.36140662],  
 [33. , 17.23368794],  
 [38.5 , 20.10596926],  
 [44. , 22.97825059],  
 [49.5 , 25.85053191],  
 [55. , 28.72281323]])

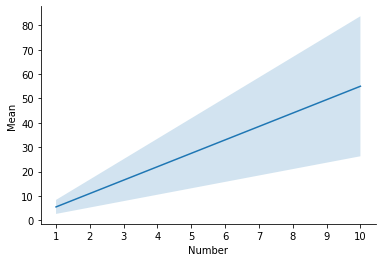
To get the number of rows in the numpy array, the shape attribute is used which return a tuple having number of rows and columns. The number of rows in the array is then passed to the range function to be used as data for the x-axis. For y-axis values and error bars, first and second columns of the array are used, respectively.

plt.errorbar(range(data\_csv.shape[0]), data\_csv[:,0], yerr=data\_csv[:,1], label="Standard deviation", fmt="o")  
plt.gca().xaxis.set\_ticks(range(data\_csv.shape[0]))  
plt.xlabel("Number")  
plt.ylabel("Mean")  
plt.show()  
plt.show()

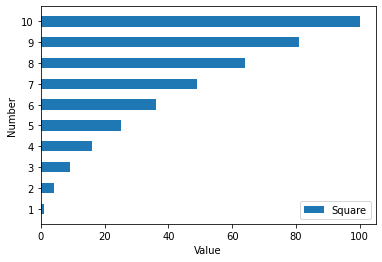


The fill\_between function can be used to plot standard deviation as a shaded region instead of error bars. This function takes two *y* values corresponding to the upper and lower bounds of the region to be filled. Given our data, we’ll create y\_upper and y\_lower arrays by adding and subtracting standard deviation from the mean, respectively. The alpha argument adjusts the transparency of the shaded region and takes a value between 0 (full transparency) and 1 (no transparency).

y\_upper = np.mean(table\_10, axis=0) + np.std(table\_10, axis=0)  
y\_lower = np.mean(table\_10, axis=0) - np.std(table\_10, axis=0)  
plt.plot(num1[0], np.mean(table\_10, axis=0))  
plt.fill\_between(num1[0], y\_upper, y\_lower, alpha=0.2)  
plt.xlabel("Number")  
plt.ylabel("Mean")  
ax = plt.gca()  
ax.xaxis.set\_ticks(num1[0])  
ax.spines[["top","right"]].set\_visible(False)  
plt.show()



Quiz: Write a program to make a graph as shown below.

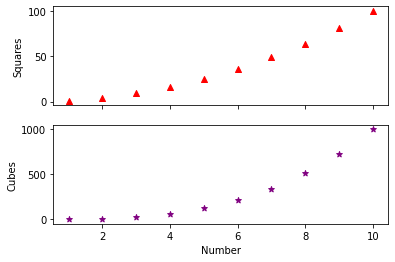


y = [a\*\*2 for a in x]  
plt.barh(x,y, height=0.5)  
plt.xlabel("Value")  
plt.ylabel("Number")  
plt.yticks(range(1,11))  
plt.legend(["Square"], loc="lower right")  
plt.show()

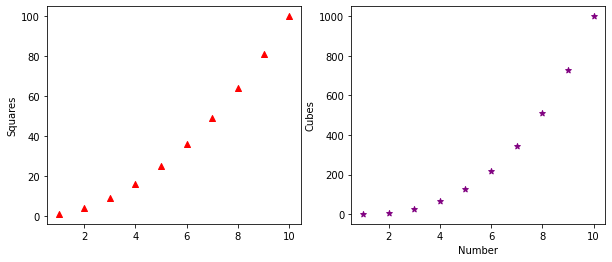
## 12.4 subplots

Many a time we need to plot multiple graphs in one figure for effecient visual analysis. In Matplotlib terminilogy these plots are refered to as subplots. The pyplot class has subplots function that return a figure and and axes object. These can be used to access and manipulated different elements of the graph. In addition, subplots can take as argument the total number of plots to create a figure. This function has a keyword argument figsize to specify the size of the plot.

x = range(1,11)  
y = [a\*\*2 for a in x]  
z = [a\*\*3 for a in x]  
  
fig, ax = plt.subplots(2,1,sharex=True) #two rows and one column  
  
ax[0].scatter(x,y,marker="^", color="red")  
ax[1].scatter(x,z,marker="\*", color="purple")  
ax[1].set\_xlabel("Number")  
ax[0].set\_ylabel("Squares")  
ax[1].set\_ylabel("Cubes")  
plt.show()

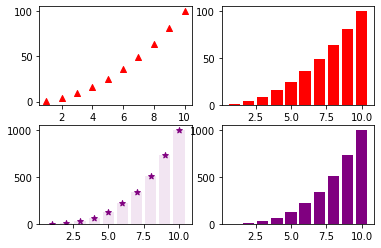


The subplots in the above figure can be stacked horizontally by changing the subplots statement to plt.subplots(1,2).



Similarly, we can create different arrangements for subplots and save an image using savefig function of the figure object. The resolution of the resulting image can be controlled using the dpi argument. The alphaargument in the code below is used to make the bars transparent.

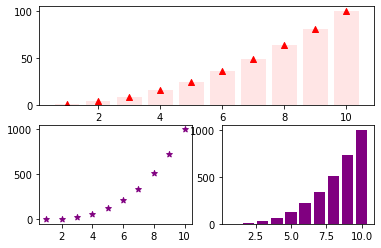
x = range(1,11)  
y = [a\*\*2 for a in x]  
z = [a\*\*3 for a in x]  
  
fig, ax = plt.subplots(2,2) #create subplots with two rows and two columns  
ax[0,0].scatter(x,y,marker="^", color="red")  
ax[0,1].bar(x,y,color="red")  
  
ax[1,0].scatter(x,z,marker="\*", color="purple")  
ax[1,0].bar(x,z,color="purple",alpha=0.1)  
ax[1,1].bar(x,z,color="purple")  
  
fig.savefig("subplot.png", dpi=300)  
plt.show()



## 12.5 subplot (without ‘s’)

The subplot() function is similar to subplots with a difference that it take an additional argument - index. This can be used to make axes span multiple columns within the subplot.

x = range(1,11)  
y = [a\*\*2 for a in x]  
z = [a\*\*3 for a in x]  
  
ax1 = plt.subplot(2,1,1)   
ax2 = plt.subplot(2,2,3)  
ax3 = plt.subplot(2,2,4)  
  
ax1.scatter(x,y,marker="^", color="red")  
ax1.bar(x,y, color="red",alpha=0.1)  
ax2.scatter(x,z,marker="\*", color="purple")  
ax3.bar(x,z,color="purple")  
  
plt.savefig("subplots2.png",dpi=300)  
plt.show()



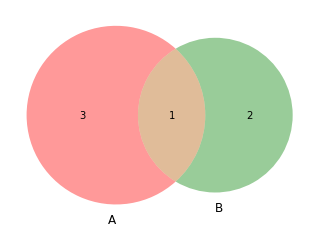
## 12.6 Venn diagrams

When there are multiple set of items, we might want to analyse the logical relationship between the sets. For instance, how many items are common between two or more sets or how many items are uniquely present in a particular set. Venn diagrams are ideally suited to visualize such logical relations across sets. The matplotlib-venn library provides functions to plot Venn diagrams for two or three sets. It can be installed using pip install matplotlib-venn command.

import matplotlib.pyplot as plt  
from matplotlib\_venn import venn2

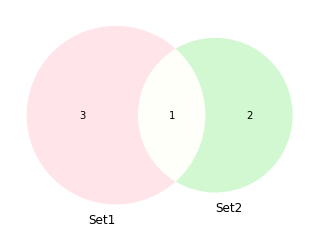
The venn2 function takes a list having two sets are an argument and plots the two-circle Venn diagram. The size of each set is in proportion to the items in that set. As discussed in the [chapter](Tuples.html) on sets, a set contains only unique values i.e. if there are duplicates then those are removed.

venn2([set(['A', 'B', 'C', 'D']), set(['D', 'E', 'F'])])  
plt.show()



The fill color can be changed using the set\_colors keyword argument and the set\_labels argument adds the labels for each set. Both these arguments take a set of two values as a valid input.

venn2([set(['A', 'B', 'C', 'D']), set(['D', 'E', 'F'])], set\_colors=('pink', 'lightgreen'),\  
 set\_labels=('Set1', 'Set2'))  
plt.show()

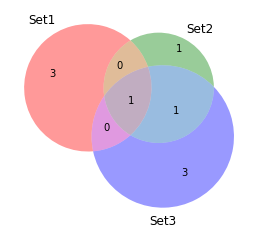


Similarly, venn3 is used for plotting three sets.

from matplotlib\_venn import venn3

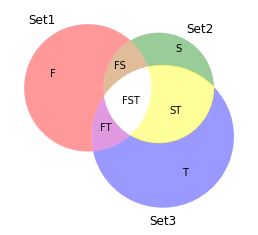
set1 = set(['A', 'B', 'C', 'D'])  
set2 = set(['D', 'E', 'F'])  
set3 = set(['D','F','G', 'H', 'I'])  
venn3([set1, set2, set3], set\_labels=('Set1', 'Set2', 'Set3'))  
plt.show()

{'A', 'C', 'D', 'B'}



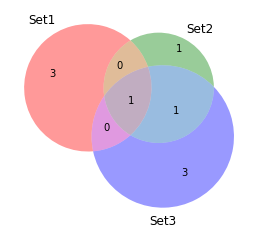
We can further customize the individual components within these plots using the get\_patch\_by\_id and get\_label\_by\_id attributes for the Venn diagram object.

v = venn3([set1, set2, set3], set\_labels=('Set1', 'Set2', 'Set3'))  
v.get\_label\_by\_id('100').set\_text('F') # label for First set  
v.get\_label\_by\_id('010').set\_text('S') # label for Second set  
v.get\_label\_by\_id('001').set\_text('T') # label for Third set  
v.get\_label\_by\_id('110').set\_text('FS') # label for intersection of First and Second   
v.get\_label\_by\_id('011').set\_text('ST') # label for intersection of Second and Third  
v.get\_label\_by\_id('101').set\_text('FT') # label for intersection of First and Third  
v.get\_label\_by\_id('111').set\_text('FST') # label for intersection of all three  
v.get\_patch\_by\_id('111').set\_color('white') # color for intersection of all three  
v.get\_patch\_by\_id('011').set\_color('yellow') # color for intersection of Second and Third



An alternate approach to plot a Venn diagram is to specify the values for the subsets within the Venn diagram instead of giving three sets. As apparent from the output of the venn3 function above, there are a total of seven values that makes up the diagram. Using the subsets keyword argument for the venn3 function we make a similar plot without give a list of sets as an argument.

venn3(subsets=(3, 1, 0, 3, 0, 1, 1), set\_labels=('Set1', 'Set2', 'Set3'))  
plt.show()



# 13. Pandas

Pandas is a library that is useful for working with different types of datasets. A dataframe is a pandas object that has a variety of function to process, analyze, and visualize data. It arranges the data in rows and columns. There are multiple ways in which a dataframe can be created. For example, a dictionary with list as values can be converted to a dataframe such that the keys become headers and values (list) are entries in the dataframe. The orientation of the dataframe by default is columns ie keys are considered as column header and values are rows. This behaviour can be changed using the orient argument. When orientation is index, an addtional argument columns can be used to specify column headers.

import pandas as pd

input\_dict = {'Column1': ['A','B','C','D','E'], \  
 'Column2':[1,2,3,4,5]}  
df1 = pd.DataFrame.from\_dict(input\_dict)  
df2 = pd.DataFrame.from\_dict(input\_dict, orient='index', columns=['Val1','Val2','Val3','Val4','Val5',])  
display(df1)  
display(df2)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | Column1 | Column2 | | --- | --- | --- | | 0 | A | 1 | | 1 | B | 2 | | 2 | C | 3 | | 3 | D | 4 | | 4 | E | 5 | | |  | Val1 | Val2 | Val3 | Val4 | Val5 | | --- | --- | --- | --- | --- | --- | | Column1 | A | B | C | D | E | | Column2 | 1 | 2 | 3 | 4 | 5 | |

print(df2.columns)

Index(['Val1', 'Val2', 'Val3', 'Val4', 'Val5'], dtype='object')

## 13.1 Indexing and Selecting

The data in a dataframes are arranged in rows and columns both of which have labels. The get a slice of a dataframe, these labels can used with the loc attribute. The columns and index attributes for a dataframe return an iteratable collection of column labels and index labels. To select a subset of a dataframe using indcies for the rows and columns use iloc. Both loc and iloc takes a slice of row label/range and column label/range. Note that a one-dimensional slice of a dataframe returns a Series object i.e. if we select one column or one row, a Series object would be returned. To convert a series object to a dataframe use to\_frame function.

Multi-indexing is also possible i.e., a row or a column can have more than one label. In such cases, selections can be made by specifing labels as a tuple.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | Column1 | Column2 | | --- | --- | --- | | 0 | A | 1 | | 1 | B | 2 | | 2 | C | 3 | | 3 | D | 4 | | 4 | E | 5 | | df1.columns:   Index(['Column1', 'Column2'], dtype='object')   df1.index:   RangeIndex(start=0, stop=5, step=1) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | Column1 | | --- | --- | | 0 | A | | 1 | B | | 2 | C | | 3 | D | | 4 | E | | |  | Column1 | | --- | --- | | 0 | A | | 1 | B | | 2 | C | | 3 | D | | 4 | E | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | Column1 | Column2 | | --- | --- | --- | | 2 | C | 3 | | 3 | D | 4 | | 4 | E | 5 | | |  | Column1 | Column2 | | --- | --- | --- | | 2 | C | 3 | | 3 | D | 4 | | 4 | E | 5 | |

## 13.2 Dataframe information

Pandas offers set of commands to get some basic information about the content of dataframes. Below are some of these command along with their corresponding output.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | Column1 | Column2 | | --- | --- | --- | | 0 | A | 1 | | 1 | B | 2 | | 2 | C | 3 | | 3 | D | 4 | | 4 | E | 5 | | Information about this dataframe  df1.shape (5, 2) df1.size 10 df1.ndim 2 df1.values [['A' 1]  ['B' 2]  ['C' 3]  ['D' 4]  ['E' 5]] |

## 13.3 Data from (and to) a csv file

The read\_csv() function can be used to create a dataframe from a csv file. To use one of the columns as indices for the dataframe add the index\_col keyword attribute.

# %load test.csv  
Name,Age,Country  
Sohan,22,India  
Sam,21,USA

df3 = pd.read\_csv("test.csv")  
df4 = pd.read\_csv("test.csv", index\_col="Country")  
display(df3)  
display(df4)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | Name | Age | Country | | --- | --- | --- | --- | | 0 | Sohan | 22 | India | | 1 | Sam | 21 | USA | | |  | Name | Age | | --- | --- | --- | | Country |  |  | | India | Sohan | 22 | | USA | Sam | 21 | |

To write a dataframe to a csv file use to\_csv function. The default delimiter is , and can be changed with the sep argument.

df4.to\_csv("output.csv")

# %load output.csv  
Country,Name,Age  
India,Sohan,22  
USA,Sam,21

## 13.4 Combining dataframes

Pandas offers a variety of way to combine dataframes. These methods differ in the options in which the data from two or more dataframes can be combined. Having a good understanding of these functions is vital of effectively working with diverse datasets.

### Concat

Pandas has a concat function that takes a list of dataframes and returns a combined dataframe. The axis argument indicates whether the dataframes would be combined across rows or columns. The default is axis=0 i.e. the dataframes are concatenated row-wise. The key argument can be used to add an additional index refering to each of the dataframe that is concatenated.

import numpy as np  
df\_1 = pd.DataFrame(np.random.uniform(1,2,size=(5, 4)), columns=list('ABCD'))  
df\_2 = pd.DataFrame(np.random.uniform(2,3,size=(5, 4)), columns=list('ABCD'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | A | B | C | D | | --- | --- | --- | --- | --- | | 0 | 1.710289 | 1.260774 | 1.047939 | 1.342935 | | 1 | 1.539253 | 1.610440 | 1.608759 | 1.518317 | | 2 | 1.471803 | 1.817016 | 1.757089 | 1.957877 | | 3 | 1.621033 | 1.152500 | 1.237414 | 1.457109 | | 4 | 1.671551 | 1.689427 | 1.698675 | 1.270451 | | |  | A | B | C | D | | --- | --- | --- | --- | --- | | 0 | 2.511460 | 2.394432 | 2.979061 | 2.699233 | | 1 | 2.431831 | 2.324515 | 2.433901 | 2.839483 | | 2 | 2.507918 | 2.410554 | 2.192603 | 2.818681 | | 3 | 2.969286 | 2.182390 | 2.412281 | 2.181399 | | 4 | 2.605576 | 2.750244 | 2.238079 | 2.926454 | |

df\_new = pd.concat([df\_1,df\_2],ignore\_index=True)  
display(df\_new)

|  | A | B | C | D |
| --- | --- | --- | --- | --- |
| 0 | 1.710289 | 1.260774 | 1.047939 | 1.342935 |
| 1 | 1.539253 | 1.610440 | 1.608759 | 1.518317 |
| 2 | 1.471803 | 1.817016 | 1.757089 | 1.957877 |
| 3 | 1.621033 | 1.152500 | 1.237414 | 1.457109 |
| 4 | 1.671551 | 1.689427 | 1.698675 | 1.270451 |
| 5 | 2.511460 | 2.394432 | 2.979061 | 2.699233 |
| 6 | 2.431831 | 2.324515 | 2.433901 | 2.839483 |
| 7 | 2.507918 | 2.410554 | 2.192603 | 2.818681 |
| 8 | 2.969286 | 2.182390 | 2.412281 | 2.181399 |
| 9 | 2.605576 | 2.750244 | 2.238079 | 2.926454 |

df\_new = pd.concat([df\_1,df\_2],axis=1)  
display(df\_new)

|  | A | B | C | D | A | B | C | D |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1.710289 | 1.260774 | 1.047939 | 1.342935 | 2.511460 | 2.394432 | 2.979061 | 2.699233 |
| 1 | 1.539253 | 1.610440 | 1.608759 | 1.518317 | 2.431831 | 2.324515 | 2.433901 | 2.839483 |
| 2 | 1.471803 | 1.817016 | 1.757089 | 1.957877 | 2.507918 | 2.410554 | 2.192603 | 2.818681 |
| 3 | 1.621033 | 1.152500 | 1.237414 | 1.457109 | 2.969286 | 2.182390 | 2.412281 | 2.181399 |
| 4 | 1.671551 | 1.689427 | 1.698675 | 1.270451 | 2.605576 | 2.750244 | 2.238079 | 2.926454 |

df\_new = pd.concat([df\_1,df\_2],keys=["First","Second"])  
display(df\_new)

|  |  | A | B | C | D |
| --- | --- | --- | --- | --- | --- |
| First | 0 | 1.710289 | 1.260774 | 1.047939 | 1.342935 |
| 1 | 1.539253 | 1.610440 | 1.608759 | 1.518317 |
| 2 | 1.471803 | 1.817016 | 1.757089 | 1.957877 |
| 3 | 1.621033 | 1.152500 | 1.237414 | 1.457109 |
| 4 | 1.671551 | 1.689427 | 1.698675 | 1.270451 |
| Second | 0 | 2.511460 | 2.394432 | 2.979061 | 2.699233 |
| 1 | 2.431831 | 2.324515 | 2.433901 | 2.839483 |
| 2 | 2.507918 | 2.410554 | 2.192603 | 2.818681 |
| 3 | 2.969286 | 2.182390 | 2.412281 | 2.181399 |
| 4 | 2.605576 | 2.750244 | 2.238079 | 2.926454 |

display(df\_new.loc["First"])

|  | A | B | C | D |
| --- | --- | --- | --- | --- |
| 0 | 1.710289 | 1.260774 | 1.047939 | 1.342935 |
| 1 | 1.539253 | 1.610440 | 1.608759 | 1.518317 |
| 2 | 1.471803 | 1.817016 | 1.757089 | 1.957877 |
| 3 | 1.621033 | 1.152500 | 1.237414 | 1.457109 |
| 4 | 1.671551 | 1.689427 | 1.698675 | 1.270451 |

### Merge

The merge function in pandas is used combine two dataframes on one or more columns. The on argument takes name(s) of column or index levels to be merged. If on is None (default) then all the columns are considred and in case of identical columns, an intersection is performed. When on is specified and the merged dataframe results in duplicate column names then suffixes argument can be used to indicate the original datafames.

# read a csv file having following data  
# Name,Age,Country  
# Sohan,22,India  
# Sam,21,USA  
  
df3 = pd.read\_csv("test.csv")  
display(df3)  
  
# make a copy of df3  
df4 = df3.copy(deep=True)  
# add rows to df4  
df4.loc[2]=["Peter", 20, "UK"]   
df4.loc[len(df4.index)] = ["Mohan", 25, "India"]  
display(df4)  
  
# merge dataframes  
df\_merged1 = pd.merge(df3,df4)  
display(df\_merged1)  
  
df\_merged2 = pd.merge(df3,df4,on=["Country","Name"],\  
 suffixes=('\_df3', '\_df4'))  
display(df\_merged2)

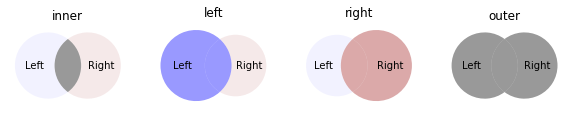
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | df3 | | | | --- | --- | --- | --- | |  | Name | Age | Country | | 0 | Sohan | 22 | India | | 1 | Sam | 21 | USA | | |  | df4 | | | | --- | --- | --- | --- | |  | Name | Age | Country | | 0 | Sohan | 22 | India | | 1 | Sam | 21 | USA | | 2 | Peter | 20 | UK | | 3 | Mohan | 25 | India | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | df\_merged1 | | | | --- | --- | --- | --- | |  | Name | Age | Country | | 0 | Sohan | 22 | India | | 1 | Sam | 21 | USA | | |  | df\_merged2 | | | | | --- | --- | --- | --- | --- | |  | Name | Age\_df3 | Country | Age\_df4 | | 0 | Sohan | 22 | India | 22 | | 1 | Sam | 21 | USA | 21 | |

### Join

The join function for a dataframe oject is used to combine one or more dataframes. By default the joining is performed along the indcies and the lsuffix and rsuffix arguments can be used to modify column names. In this case, new columns from one of the dataframes are added to the other. The on keyword specifies the index label on which to join the dataframes.

The joining of dataframe can be performed with different logics in terms of how to handle the overlapping columns. The diagram below shows the pictorial representation of four options that we have. In these Venn diagrams the darker shade shows the subset that will be selected. Here left refers to the dataframe on which the join function is called and right is the dataframe which goes as the argument for the join function.



display(df3.join(df4,lsuffix='\_df3', rsuffix='\_df4'))

|  | Name\_df3 | Age\_df3 | Country\_df3 | Name\_df4 | Age\_df4 | Country\_df4 |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | Sohan | 22 | India | Sohan | 22 | India |
| 1 | Sam | 21 | USA | Sam | 21 | USA |

display(df3.set\_index("Country"))

|  | Name | Age |
| --- | --- | --- |
| Country |  |  |
| India | Sohan | 22 |
| USA | Sam | 21 |

To join df4 to df3 on Country, we need to set the index of df4 to Country. Note that there are two rows with index 0 because there is one row with India in df3 and two such rows in df\_4. Also, in the joined dataframe there is no data for UK since there was no row in df3 with UK in Country column. This behaviour can be changed by adding the how argument to join function (see below).

display(df3.join(df4.set\_index("Country"),on="Country", lsuffix='\_df3', rsuffix='\_df4'))

|  | Name\_df3 | Age\_df3 | Country | Name\_df4 | Age\_df4 |
| --- | --- | --- | --- | --- | --- |
| 0 | Sohan | 22 | India | Sohan | 22 |
| 0 | Sohan | 22 | India | Mohan | 25 |
| 1 | Sam | 21 | USA | Sam | 21 |

To get the joined dataframe with Country as index, set indcies for both the dataframes to Country.

display(df3.join(df4.set\_index("Country"),on="Country", lsuffix='\_df3', rsuffix='\_df4', how='outer'))

|  | Name\_df3 | Age\_df3 | Country | Name\_df4 | Age\_df4 |
| --- | --- | --- | --- | --- | --- |
| 0.0 | Sohan | 22.0 | India | Sohan | 22 |
| 0.0 | Sohan | 22.0 | India | Mohan | 25 |
| 1.0 | Sam | 21.0 | USA | Sam | 21 |
| NaN | NaN | NaN | UK | Peter | 20 |

display(df3.set\_index("Country").join(df4.set\_index("Country"),on="Country", lsuffix='\_df3', rsuffix='\_df4', how="outer"))

|  | Country | Name\_df3 | Age\_df3 | Name\_df4 | Age\_df4 |
| --- | --- | --- | --- | --- | --- |
| India | India | Sohan | 22.0 | Sohan | 22 |
| India | India | Sohan | 22.0 | Mohan | 25 |
| USA | USA | Sam | 21.0 | Sam | 21 |
| NaN | UK | NaN | NaN | Peter | 20 |

## 13.5 Groupby

We can create groups for same values in a column to apply a function to all rows having a particular value.

students = [["Sam","Peter","Mohan", "Mike"], ["UG","PG","UG","PG"], [70,80,90,70]]  
df\_students = pd.DataFrame(students).T  
  
df\_students.columns=["Name","Program","Marks"]  
display(df\_students)

|  | Name | Program | Marks |
| --- | --- | --- | --- |
| 0 | Sam | UG | 70 |
| 1 | Peter | PG | 80 |
| 2 | Mohan | UG | 90 |
| 3 | Mike | PG | 70 |

df\_students.set\_index("Program", inplace=True)  
display(df\_students)

|  | Name | Marks |
| --- | --- | --- |
| Program |  |  |
| UG | Sam | 70 |
| PG | Peter | 80 |
| UG | Mohan | 90 |
| PG | Mike | 70 |

df\_students.groupby(level="Program")["Marks"].mean()

Program  
PG 75.0  
UG 80.0  
Name: Marks, dtype: float64

## 13.6 Styling

The pandas dataframe has a style attribute that return a styler object which has different functions to customize dataframe display. E.g., to highlight minimun and maximum values in a dataframe, the hightlight\_min and highlight\_max functions for the styler object can be used. The min and max values can be selected column-wise or row-wise by setting the axis argument to 0 or 1, respectively. To highlight min or max value for the entire dataframe use axis = None. The props argument is used to set the highlight properties and takes a valid CSS (Cascading Style Sheet) property.

df\_1 = pd.DataFrame(np.random.uniform(1,2,size=(5, 4)), columns=list('ABCD'))  
display(df\_1.style.highlight\_max()\  
 .set\_caption("Column-wise highlighting of maximum value"))  
display(df\_1.style.highlight\_min(axis=1,props="font-weight:bold;background-color:pink;")\  
 .set\_caption("Row-wise highlighting of minimum value"))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Table 13.1: Column-wise highlighting of maximum value   |  | A | B | C | D | | --- | --- | --- | --- | --- | | 0 | 1.036642 | 1.265409 | 1.593481 | 1.470786 | | 1 | 1.169744 | 1.747707 | 1.411616 | 1.924018 | | 2 | 1.620145 | 1.927595 | 1.532973 | 1.063995 | | 3 | 1.895642 | 1.669849 | 1.834086 | 1.254635 | | 4 | 1.831751 | 1.220100 | 1.203569 | 1.954389 | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Table 13.2: Row-wise highlighting of minimum value   |  | A | B | C | D | | --- | --- | --- | --- | --- | | 0 | 1.036642 | 1.265409 | 1.593481 | 1.470786 | | 1 | 1.169744 | 1.747707 | 1.411616 | 1.924018 | | 2 | 1.620145 | 1.927595 | 1.532973 | 1.063995 | | 3 | 1.895642 | 1.669849 | 1.834086 | 1.254635 | | 4 | 1.831751 | 1.220100 | 1.203569 | 1.954389 | | |

We can also highlight all the values in a dataframe with color gradient. The coloring can be applied to the text or background. The cmap argument for background\_gradient and text\_gradient functions can be used to customize the coloring based on a colormap.

display(df\_1.style.background\_gradient()\  
 .set\_caption("Column-wise background gradient")\  
 .set\_properties(\*\*{"font-weight":"bold", "padding":"1em"})\  
 .format("{:.2f}"))  
display(df\_1.style.text\_gradient(axis=1, cmap="winter")\  
 .set\_caption("Row-wise text gradient")\  
 .set\_properties(\*\*{"font-weight":"bold", "padding":"1em"})\  
 .format("{:.2f}"))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Table 13.3: Column-wise background gradient   |  | A | B | C | D | | --- | --- | --- | --- | --- | | 0 | 1.04 | 1.27 | 1.59 | 1.47 | | 1 | 1.17 | 1.75 | 1.41 | 1.92 | | 2 | 1.62 | 1.93 | 1.53 | 1.06 | | 3 | 1.90 | 1.67 | 1.83 | 1.25 | | 4 | 1.83 | 1.22 | 1.20 | 1.95 | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Table 13.4: Row-wise text gradient   |  | A | B | C | D | | --- | --- | --- | --- | --- | | 0 | 1.04 | 1.27 | 1.59 | 1.47 | | 1 | 1.17 | 1.75 | 1.41 | 1.92 | | 2 | 1.62 | 1.93 | 1.53 | 1.06 | | 3 | 1.90 | 1.67 | 1.83 | 1.25 | | 4 | 1.83 | 1.22 | 1.20 | 1.95 | | |

### Bar charts within dataframe

The values in a dataframe can be decorated with bar plots using the bar function for the styler object. In this example, we’ll display the dataframe after applying the following customizations:  
- bar plots with custom color map.  
- format the floats to display upto two decimal places.  
- make the text bold.  
- center align the text in the dataframe.  
- hide the indcies.

df\_bar = pd.DataFrame(np.random.uniform(-10,10,size=(5, 4)), columns=list('ABCD'))  
from matplotlib.colors import ListedColormap  
cmp = ListedColormap(["pink","lightblue"])  
display(df\_bar.style  
 .bar(vmin=-10, vmax=10, align=0, cmap=cmp)\  
 .format("{:.2f}")\  
 .set\_properties(\*\*{"font-weight":"bold", "text-align":"center"})\  
 .set\_table\_styles([dict(selector='th', props=[('text-align', 'center')])])\  
 .hide(axis="index")\  
 .set\_caption("Dataframe with bar charts"))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table 13.5: Dataframe with bar charts   | A | B | C | D | | --- | --- | --- | --- | | -4.55 | -7.13 | -9.91 | -2.22 | | -6.40 | 5.80 | -4.14 | 3.11 | | -4.75 | 8.02 | -0.00 | 3.65 | | -6.30 | -8.01 | 2.33 | -0.02 | | -9.94 | 5.72 | 6.49 | 6.30 | |

## 13.7 Ploting

Dataframe has a plot() function to do basic visualization. The kind attribute for this function can be used to change the plot type.

df\_col1 = pd.DataFrame(np.array(range(1,6))\*\*2)  
df\_col2 = pd.DataFrame(np.array(range(1,6))\*\*3)  
  
df\_comb = pd.concat([df\_col1,df\_col2], axis=1, ignore\_index=True)  
df\_comb.columns = ["Squares", "Cubes"]  
df\_comb.index = range(1,6)  
display(df\_comb)

|  | Squares | Cubes |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 4 | 8 |
| 3 | 9 | 27 |
| 4 | 16 | 64 |
| 5 | 25 | 125 |

plot1 = df\_comb.plot(title="Line Plot")  
plot2 = df\_comb.plot(kind="bar", title="Bar Plot")

|  |  |
| --- | --- |
|  |  |

### The iris dataset

One of the popular datasets for learning data analysis is the iris dataset which has data for four iris flower features (sepal length, sepal width, petal length, and petal width). These data are there for the three species namely *Iris setosa*, *Iris versicolour*, and *Iris virginica*. Let’s import this data create a dataframe and practice data visualization.

csv\_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'  
# using the attribute information as the column names  
col\_names = ['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width','Class']  
iris = pd.read\_csv(csv\_url, names = col\_names)

display(iris)  
print(iris.dtypes)

|  | Sepal\_Length | Sepal\_Width | Petal\_Length | Petal\_Width | Class |
| --- | --- | --- | --- | --- | --- |
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |
| ... | ... | ... | ... | ... | ... |
| 145 | 6.7 | 3.0 | 5.2 | 2.3 | Iris-virginica |
| 146 | 6.3 | 2.5 | 5.0 | 1.9 | Iris-virginica |
| 147 | 6.5 | 3.0 | 5.2 | 2.0 | Iris-virginica |
| 148 | 6.2 | 3.4 | 5.4 | 2.3 | Iris-virginica |
| 149 | 5.9 | 3.0 | 5.1 | 1.8 | Iris-virginica |

Sepal\_Length float64  
Sepal\_Width float64  
Petal\_Length float64  
Petal\_Width float64  
Class object  
dtype: object

To get the summary statistics for this dataset we’ll use the describe function for the iris dataframe. This function calculates different statistical parameters for all the columns with numeric data. Notice that this summary would be for the entire data and so might not be of much use. A better approach would be first aggreate the data by species so that we can get summary statistics for each species.

iris.describe()

|  | Sepal\_Length | Sepal\_Width | Petal\_Length | Petal\_Width |
| --- | --- | --- | --- | --- |
| count | 150.000000 | 150.000000 | 150.000000 | 150.000000 |
| mean | 5.843333 | 3.054000 | 3.758667 | 1.198667 |
| std | 0.828066 | 0.433594 | 1.764420 | 0.763161 |
| min | 4.300000 | 2.000000 | 1.000000 | 0.100000 |
| 25% | 5.100000 | 2.800000 | 1.600000 | 0.300000 |
| 50% | 5.800000 | 3.000000 | 4.350000 | 1.300000 |
| 75% | 6.400000 | 3.300000 | 5.100000 | 1.800000 |
| max | 7.900000 | 4.400000 | 6.900000 | 2.500000 |

iris\_grp = iris.groupby(by="Class")  
iris\_grp.describe()['Petal\_Length']

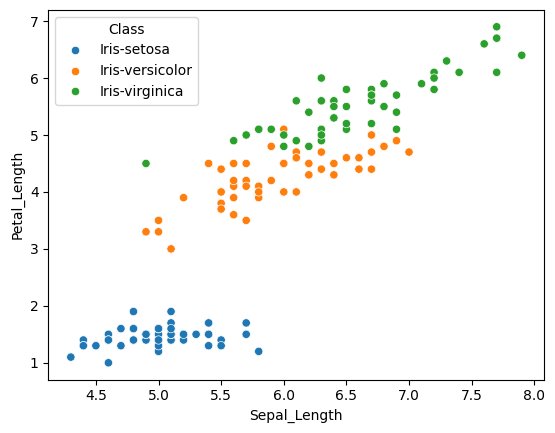
|  | count | mean | std | min | 25% | 50% | 75% | max |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Class |  |  |  |  |  |  |  |  |
| Iris-setosa | 50.0 | 1.464 | 0.173511 | 1.0 | 1.4 | 1.50 | 1.575 | 1.9 |
| Iris-versicolor | 50.0 | 4.260 | 0.469911 | 3.0 | 4.0 | 4.35 | 4.600 | 5.1 |
| Iris-virginica | 50.0 | 5.552 | 0.551895 | 4.5 | 5.1 | 5.55 | 5.875 | 6.9 |

### Seaborn library

The Seaborn library is data visualization library based on Matplotlib. This library is particularly suitable for ploting with dataframes. The libaray has been specifically designed and develop in way to focus on the data analysis rather than worrying about nuances of drawing plot.

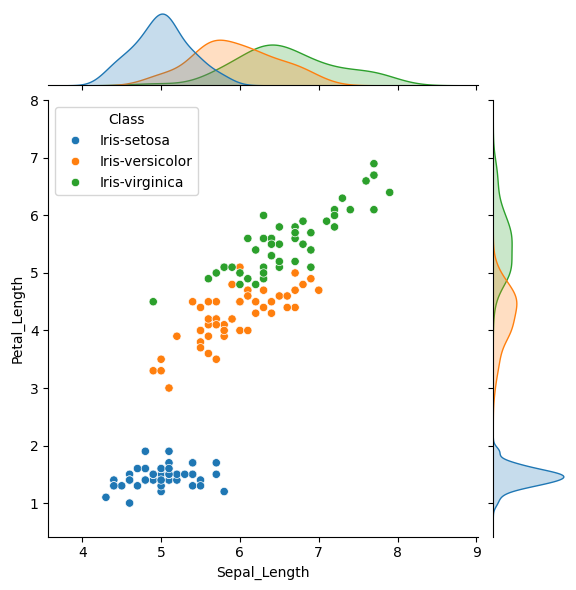
import matplotlib.pyplot as plt  
import seaborn as sns

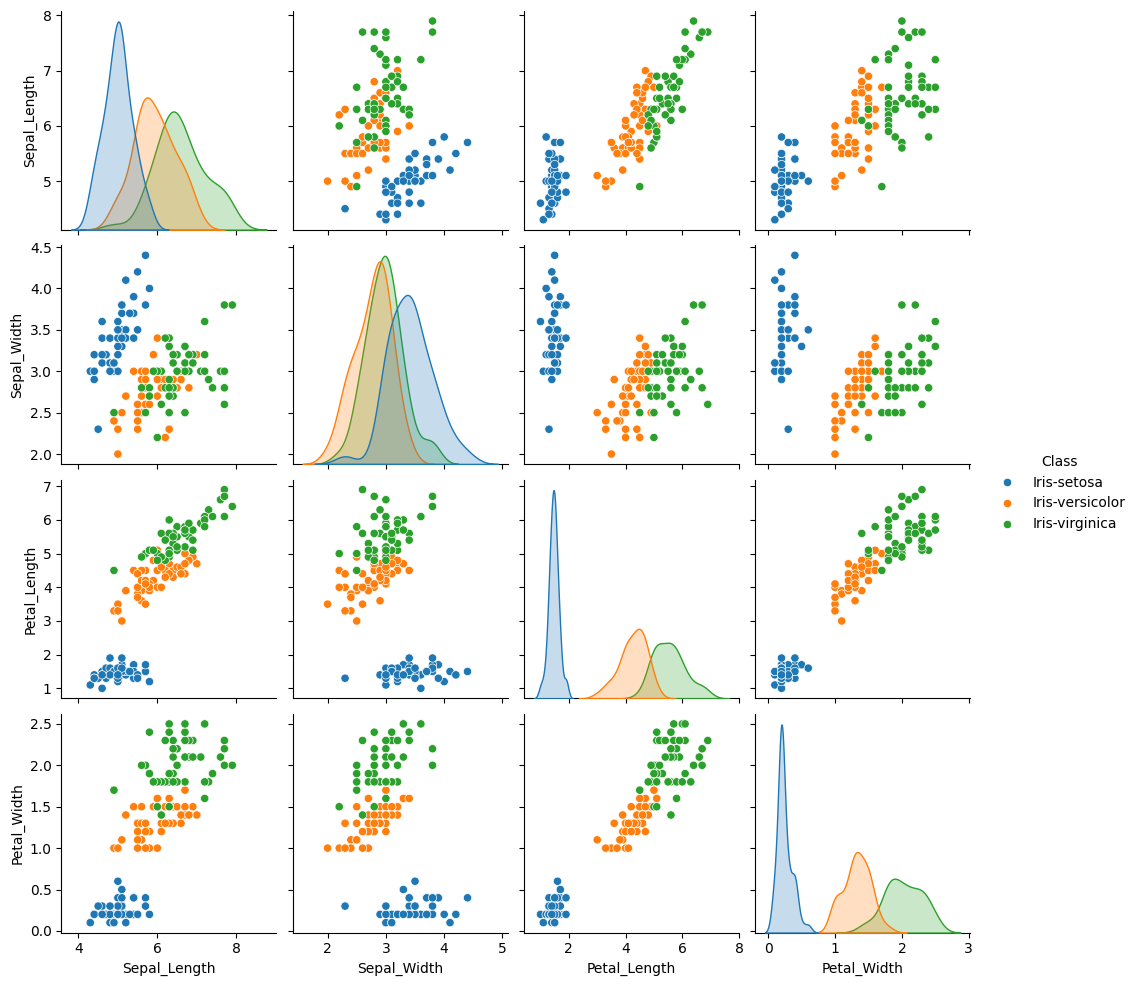
p1 = sns.scatterplot(data=iris,x="Sepal\_Length",y="Petal\_Length",hue="Class")



The seaborn library has dedicated functions to facilitate ploting informative visualizations. E.g., jointplot is used for rendering a scatter plot along with the corresponding data distributions. Similarly, pairplot can be used to render pairwise plots for all the numeric data in a dataframe.

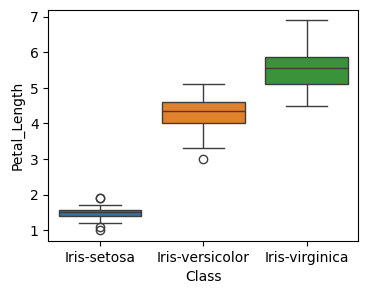
plot1 = sns.jointplot(data=iris,x="Sepal\_Length",y="Petal\_Length",hue="Class")  
plot2 = sns.pairplot(data=iris,hue="Class")





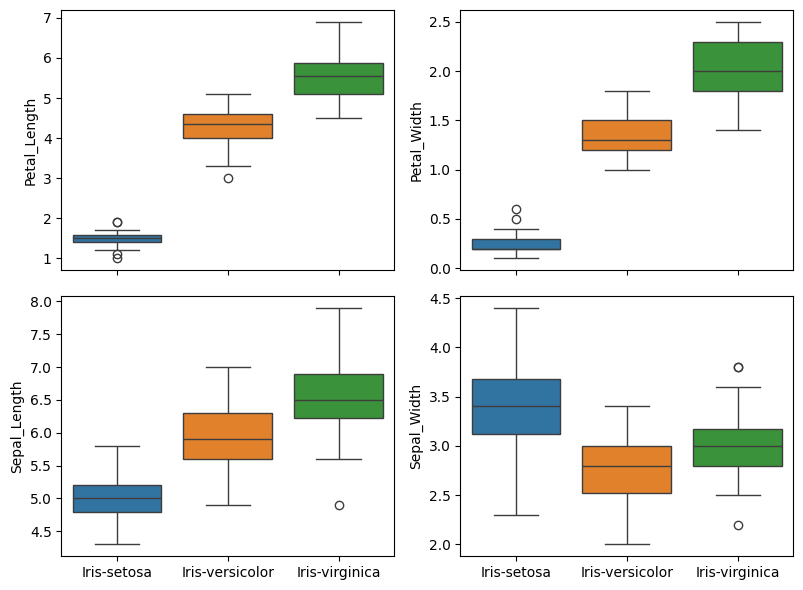
We can also visualize statistical information graphically, e.g. in the form of a boxplot. For instance, to see the distribution of the values for Petal\_Length across the three iris species, we can make a boxplot as follows.

fig, ax = plt.subplots(figsize=(4,3))  
sns.boxplot(data=iris,x="Class", y="Petal\_Length", hue="Class", ax=ax)  
plt.show()



Similarly, we can have separate boxplots for all four features as a grid.

fig, ax = plt.subplots(2, 2, figsize=(8,6), sharex=True)  
sns.boxplot(data=iris,x="Class", y="Petal\_Length", hue="Class", ax=ax[0,0]).set(xlabel='')  
sns.boxplot(data=iris,x="Class", y="Petal\_Width", hue="Class", ax=ax[0,1]).set(xlabel='')  
sns.boxplot(data=iris,x="Class", y="Sepal\_Length", hue="Class", ax=ax[1,0]).set(xlabel='')  
sns.boxplot(data=iris,x="Class", y="Sepal\_Width", hue="Class", ax=ax[1,1]).set(xlabel='')  
plt.tight\_layout()  
plt.show()

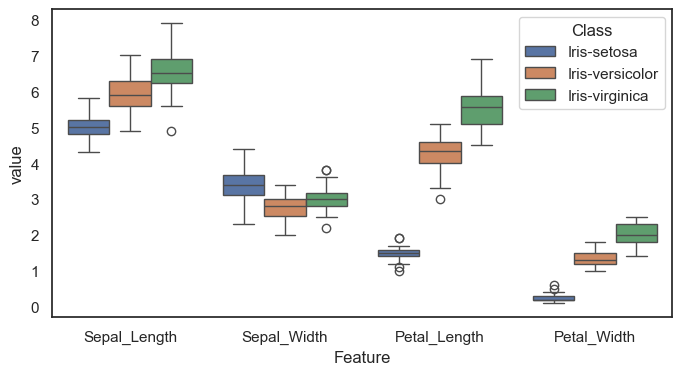


Now, let’s say we want to have the boxplot plots for each of the feature grouped together i.e. a graph that shows four sets of boxplots with groups colored according to the species name.  
For this kind of a plot, we first need to modify (or reformat) our dataframe so that its in the required format. Using the melt function in pandas, the original iris dataframe would be modified such that the values for all four flower features appear one per row. This would allow us to perform statistical calculations (required for boxplot) on all the feature and color the plot based on the species. Notice that the original iris dataframe’s shape is 150 X 5 (which implies a total of 600 datapoints since there are 4 data columns) while the *melted* dataframe’s shape is 600 X 3.

iris\_melt = pd.melt(iris, id\_vars=['Class'],\  
 value\_vars=["Sepal\_Length","Sepal\_Width","Petal\_Length","Petal\_Width"],\  
 var\_name='Feature')  
iris\_melt

|  | Class | Feature | value |
| --- | --- | --- | --- |
| 0 | Iris-setosa | Sepal\_Length | 5.1 |
| 1 | Iris-setosa | Sepal\_Length | 4.9 |
| 2 | Iris-setosa | Sepal\_Length | 4.7 |
| 3 | Iris-setosa | Sepal\_Length | 4.6 |
| 4 | Iris-setosa | Sepal\_Length | 5.0 |
| ... | ... | ... | ... |
| 595 | Iris-virginica | Petal\_Width | 2.3 |
| 596 | Iris-virginica | Petal\_Width | 1.9 |
| 597 | Iris-virginica | Petal\_Width | 2.0 |
| 598 | Iris-virginica | Petal\_Width | 2.3 |
| 599 | Iris-virginica | Petal\_Width | 1.8 |

fig, ax = plt.subplots(figsize=(8,4))  
plot3 = sns.boxplot(data=iris\_melt,x="Feature", y="value",hue="Class", ax=ax)



The catplot function provides an intutive interface to visually analyze the relationship between numerical and categorical data. There are three different options for plot layout – scatter plot or distribution plot or estimate plot – each of which points to an axis-level plot function. The plot selection is made using the kind keyword argument. The table below shows the available options for the kind argument and their corresponding axis-level functions. The size of the plot can be adjusted using the height and aspect keyword arguments.

Categorical plot

kind

Axis-level function

Scatter plot

strip (default)

stripplot()

swarm

swarmplot()

Distribution plot

box

boxplot()

violin

violinplot()

boxen

boxenplot()

Estimate plot

point

pointplot()

bar

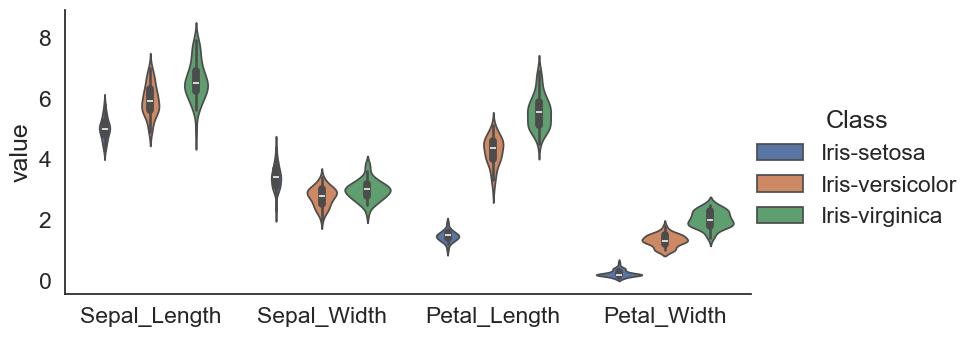
barplot()

count

countplot()

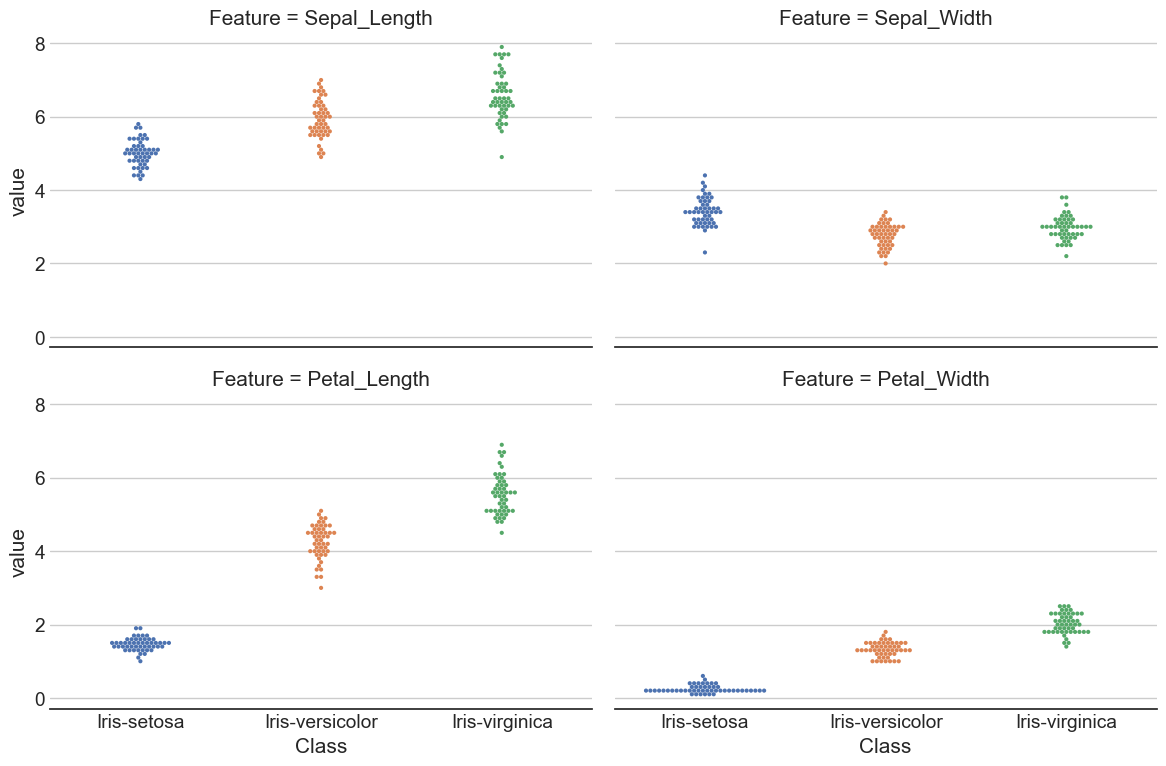
The code below creates a violin plot using the above data. The x-axis label is removed using the set function for the FacetGrid object and the font size for all the labels is increase using the set\_theme function from sns.

sns.set\_theme(style="white", font\_scale=1.5)  
p2 = sns.catplot(data=iris\_melt, x="Feature", y="value", kind="violin",   
 hue="Class", height=4, aspect=2)  
p2.set(xlabel=None)  
plt.show()



Similarly, we can make a swarm plot for each feature and color it by species. A swarm plot is like a scatterplot such that the position of the points are to avoid overlap. The catplot function returns a FacetGrid which can be customized to render desired representation. For our data there will be four plots in the facet grid which will be arranged as one row and four columns by default. The code below, makes a feature-wise swarm plot with two columns (col\_warp=2) in the Facetgrid. The final plot has has two enhancements — 1) the veritcal line for the y-axis is removed and 2) the tick marks removed from both the axis and major grid lines shown for y-axis.

sns.set\_theme(style="white", font\_scale=1.25)  
p2 = sns.catplot(data=iris\_melt, x="Class", y="value", kind="swarm", size=3,  
 col="Feature", height=4, aspect=1.5, col\_wrap=2, hue="Class", errorbar=None)  
p2.despine(left=True)  
for ax in p2.axes.flat:  
 ax.tick\_params(axis='both', which='both', length=0)  
 ax.yaxis.grid(True)



# 14. Functions

In programming when a sub-set of code is to be execute in one go, we have an option to give a name to that sub-set and that name is now a function. As an analogy consider the steps in making tea.

**Make Tea**  
- Boil water  
- Add tea  
- Add sugar  
- Add milk

**Make Tea with options**  
- Boil water  
- Add tea  
- Add sugar  
- **If** black tea skip this step  
  **else** Add milk

The above steps need to be done every time we make tea. Also, we can create a new sub-set with some customization — “Make Tea with options”. So, whenever we say “make tea” it is implied that the step mentioned above would be performed and we don’t need to mention the steps again; just make tea is enough. Similarly, in programming, we can assign a keyword to a code block such that every time that keyword is called all the code within that block gets executed. Continuing with our analogy, we now have two “functions” – “Make Tea” and “Make Tea with options”.

So far we have used various in-built Python functions. Now let’s look at how we can define our own functions to do a specific task. To **def**ine a function, def keyword is used followed by a function name and parenthesis. The def statement ends with a colon and the statement within the function are indented (a syntax similar to for and if statements). A function can be initialized with or without arguments. To call a function, specify the function name along with the parenthesis. Below is a “Hello world!” function.

def newFunction():  
 print("Hello World!")  
newFunction()

Hello World!

The argument(s) that a function takes goes inside the parenthesis when defining the function. An argument can be positional i.e. it must be specified in a particular sequence or it can be a keyword argument (these can be specified in any order). An important point to note about functions is that the variables within a function have a local scope i.e. a variable declared within a function cannot be accessed outside the function. So, let’s say we have a function that does some calculation and we would like to get the result of that calculation. In such cases, we need to *return* the variable(s) using the return keyword. When calling the function these returned values are assigned to specified variables. Below is an example of a function with two positional arguments. Notice that when calling this function the order of argument is important.

## 14.1 Arguments

def calcPower(a,b):  
 c = a\*\*b  
 return c  
output\_1 = calcPower(3,4)  
print(output\_1)  
output\_2 = calPower(4,3)  
print(output\_2)

81  
64

## 14.2 Keyword arguments

We can have the same function with keyword arguments such that now the arguments can be specified in any order. All keyword arguments need to have a default value. If a value for a keyword argument is not specified while calling the function, the default value of the keyword is used.

def calcPower(number=1,power=1):  
 c = number\*\*power  
 return c  
output\_1 = calcPower(power=3,number=4)  
print(output\_1)

64

It is important to note here that keyword arguments can be specified without keywords as well when calling a function. In this case, the order of arguments passed would be matched to the order of the keywords in the function definition.

calcPower(2,3)

8

## 14.3 Variable number of arguments

A function can have both positional and keyword arguments and in such case, the keyword arguments must be specified after positional arguments.

It is also possible to have an arbitrary number of positional and keyword arguments. The \*args argument maintains a tuple named args that stores all the arguments passed when a function is called.

def sumNums(\*args):  
 '''This function would add any set of numbers.  
 It has been defined using \*args.'''  
 c = 0  
 for num in args:  
 c = c+num  
 return c  
print(sumNums(4,5,6))

15

The tripple quoted string in the above function not only acts a comment but is also available as *help* for this function accessible via help function.

help(sumNums)

Help on function sumNums in module \_\_main\_\_:  
  
sumNums(\*args)  
 This function would add any set of numbers.  
 It has been defined using \*args.

Quiz: Write a function to calculate (a + b)2.

def squareSum(a,b):  
 c = a\*\*2 + b\*\*2 + 2\*a\*b  
 return c  
num1 = 4  
num2 = 5  
print(f"The square of sum of {num1} and {num2} is {squareSum(num1,num2)}.")

## 14.4 Variable number of keyword arguments

The \*\*kwargs argument is used to have an arbitrary number of **k**ey**w**ord **arg**ument**s**. Note the two asterisks. In this case, a kwargs dictionary is maintained which as the keyword and its value as key-value pairs of the dictionary.

def func1(\*\*kwargs):  
 all\_keywords = kwargs.keys()  
 all\_values = kwargs.values()  
 return all\_keywords, all\_values  
k1,v1 = func1(name="Sam", age=20)  
print(k1)  
print(v1)

dict\_keys(['name', 'age'])  
dict\_values(['Sam', 20])

## 14.5 Reusing functions

Function is a python file can be accessed in another python code. E.g., we have a file fileA.py that has function funcInFileA. Now to call this function from FileB.py we need to import this function — from fileA import funcInFileA.

|  |
| --- |
| Recommendation |
| To practice function import it is advisable to do it on a console rather than inside notebook. |

# fileA.py  
def funcInFileA():  
 print("Inside File A")

#fileB.py  
from fileA import funcInFileA  
funcInFileA()

Inside File A

Save the above two files in the same folder and then execute fileB.py. If we execute file fileA.py then there would be no output since we are not calling the function funcInFileA in that file. In case we do call funcInFileA in fileA.py then this function would be called twice upon calling from fileB.py.

If you would like to have an option to call funcInFileA from both the files *and* prevent calling it twice when importing this function then you need to use a special variable called \_\_name\_\_ (note the two underscores at the beginning and the end). This variable stores the name of the module which is being executed. Each python file or a module has a \_\_name\_\_ variable associated with it. When a python file is executed this variable is assigned a value \_\_main\_\_.

print(\_\_name\_\_)

\_\_main\_\_

Here the value is \_\_main\_\_ because we are executing the code directly. When we import a function then the value of the \_\_name\_\_ variable is set to the filename in which we have the imported function. So the recommended way to call funcInFileA would be to first check the value of \_\_name\_\_ followed by conditional calling. At the end of fileA.py, we can the following code.

#Add this to fileA.py  
if \_\_name\_\_ == "\_\_main\_\_":  
 funcInFileA()

Now, the function is called when we execute either of the files. Upon executing fileB.py, since the function is imported, the \_\_name\_\_ variable for fileA would not be equal to \_\_main\_\_ and hence would not be called because the if condition there would be false.

# 15. Introduction to BioPython

Python offers a variety of functions to work with text data (Strings) that in turn make it easier to work with biological data such as DNA or protein sequences. BioPython library provides a set of classes dedicated to parsing and analysis of different type of biological data. The functions avaiable in BioPython helps researcher to progammatically process the data. Below we’ll see some of the features in Biopython for working with biological data.

To install Biopython library run pip install biopython. For more details regarding Biopython installation and tutorials, please refer to the [Biopython wiki](https://biopython.org/wiki/Biopython2).

To check the version of Biopython, run the following command.

import Bio   
print(Bio.\_\_version\_\_)

1.84

## 15.1 Sequence Object

To work with sequences, we’ll need the Bio.Seq class which has the required functions for reading and writing sequence data. Once we have imported this class we can create objects having required data. The example below shows constructing a sequence object with a DNA sequence and then using the complement and translate functions to find the sequence of the complementary strand and the translated protein sequence, respectively.

from Bio.Seq import Seq  
new\_sequence = Seq('AATTGGAACCTT')  
print(new\_sequence)  
print(new\_sequence.complement())  
print(new\_sequence.translate())

AATTGGAACCTT  
TTAACCTTGGAA  
NWNL

Looking at the sequence object, one might wonder how is this different from a string object. Because we can easily initialize a string variable with the required sequence as the value. The difference here is that the two are the objects of different classes i.e. the Seq object belongs to the Bio.Seq.Seq class whereas the string object belongs to Str. These differences in turn indicate the differences in the kind of functionality associated with these objects (think object-oriented programming). For example, the complement and translate functions make sense only in the case of sequences and not in the case of any generic text.

## 15.2 The SeqUtils package

The [Bio.SeqUtils package](https://biopython.org/docs/1.75/api/Bio.SeqUtils.html) offers a set of utility functions for getting some basic information about the sequences. These include, e.g., calculating the GC content or the melting temperature for the DNA sequences.

from Bio.SeqUtils import gc\_fraction  
from Bio.SeqUtils import MeltingTemp as mt  
print(gc\_fraction(new\_sequence))  
print(mt.Tm\_Wallace(new\_sequence)) # Tm\_GC can also be used.

0.3333333333333333  
32.0

Similarly, there are functions to get information about the protein sequences. The ProtParam module has utility functions for performing basic analysis of protein sequences. For example, the molecular\_weight function returns the molecular weight of the protein and the count\_amino\_acids function returns a dictionary having the count for all the amino acids.

from Bio.SeqUtils.ProtParam import ProteinAnalysis  
P1 = ProteinAnalysis("MFAEGRNREST")  
print(P1.molecular\_weight())  
print(P1.count\_amino\_acids()) # get\_amino\_acid\_percent can also be used.

1297.3975  
{'A': 1, 'C': 0, 'D': 0, 'E': 2, 'F': 1, 'G': 1, 'H': 0, 'I': 0, 'K': 0, 'L': 0, 'M': 1, 'N': 1, 'P': 0, 'Q': 0, 'R': 2, 'S': 1, 'T': 1, 'V': 0, 'W': 0, 'Y': 0}

## 15.3 Reading and writing sequences

When we are working with the sequence data, we generally do not type sequences instead we read (and write) sequences from a file. We’ll now see how to read and write sequences using BioPython.

To create a sequence object by reading sequence from a file, we can use the SeqIO class. The parse function in this class can read and write sequences in different formats. This function take two arguments - file name and format, and return an iterator having all the sequences. The code below shows reading a file having multiple sequences and printing the sequences using the seq attribute. Note that this would print sequnces without any annotations. The description attribute of SeqIO object can used to print the description of a sequence as given in the input file. In the code below, the output would be restricted to first three sequences. If you need to print the data for all sequences then remove the if block. The sequence file used in the example below is available [here](Human_MAPK.fasta)

from Bio import SeqIO  
count=0  
for all\_seqs in SeqIO.parse("Human\_MAPK.fasta", "fasta"):  
 print(all\_seqs.description)  
 print(all\_seqs.seq)  
 count += 1  
 if(count == 3):  
 break

sp|O15264|MK13\_HUMAN Mitogen-activated protein kinase 13 OS=Homo sapiens OX=9606 GN=MAPK13 PE=1 SV=1  
MSLIRKKGFYKQDVNKTAWELPKTYVSPTHVGSGAYGSVCSAIDKRSGEKVAIKKLSRPFQSEIFAKRAYRELLLLKHMQHENVIGLLDVFTPASSLRNFYDFYLVMPFMQTDLQKIMGMEFSEEKIQYLVYQMLKGLKYIHSAGVVHRDLKPGNLAVNEDCELKILDFGLARHADAEMTGYVVTRWYRAPEVILSWMHYNQTVDIWSVGCIMAEMLTGKTLFKGKDYLDQLTQILKVTGVPGTEFVQKLNDKAAKSYIQSLPQTPRKDFTQLFPRASPQAADLLEKMLELDVDKRLTAAQALTHPFFEPFRDPEEETEAQQPFDDSLEHEKLTVDEWKQHIYKEIVNFSPIARKDSRRRSGMKL  
sp|O60271|JIP4\_HUMAN C-Jun-amino-terminal kinase-interacting protein 4 OS=Homo sapiens OX=9606 GN=SPAG9 PE=1 SV=4  
MELEDGVVYQEEPGGSGAVMSERVSGLAGSIYREFERLIGRYDEEVVKELMPLVVAVLENLDSVFAQDQEHQVELELLRDDNEQLITQYEREKALRKHAEEKFIEFEDSQEQEKKDLQTRVESLESQTRQLELKAKNYADQISRLEEREAELKKEYNALHQRHTEMIHNYMEHLERTKLHQLSGSDQLESTAHSRIRKERPISLGIFPLPAGDGLLTPDAQKGGETPGSEQWKFQELSQPRSHTSLKVSNSPEPQKAVEQEDELSDVSQGGSKATTPASTANSDVATIPTDTPLKEENEGFVKVTDAPNKSEISKHIEVQVAQETRNVSTGSAENEEKSEVQAIIESTPELDMDKDLSGYKGSSTPTKGIENKAFDRNTESLFEELSSAGSGLIGDVDEGADLLGMGREVENLILENTQLLETKNALNIVKNDLIAKVDELTCEKDVLQGELEAVKQAKLKLEEKNRELEEELRKARAEAEDARQKAKDDDDSDIPTAQRKRFTRVEMARVLMERNQYKERLMELQEAVRWTEMIRASRENPAMQEKKRSSIWQFFSRLFSSSSNTTKKPEPPVNLKYNAPTSHVTPSVKKRSSTLSQLPGDKSKAFDFLSEETEASLASRREQKREQYRQVKAHVQKEDGRVQAFGWSLPQKYKQVTNGQGENKMKNLPVPVYLRPLDEKDTSMKLWCAVGVNLSGGKTRDGGSVVGASVFYKDVAGLDTEGSKQRSASQSSLDKLDQELKEQQKELKNQEELSSLVWICTSTHSATKVLIIDAVQPGNILDSFTVCNSHVLCIASVPGARETDYPAGEDLSESGQVDKASLCGSMTSNSSAETDSLLGGITVVGCSAEGVTGAATSPSTNGASPVMDKPPEMEAENSEVDENVPTAEEATEATEGNAGSAEDTVDISQTGVYTEHVFTDPLGVQIPEDLSPVYQSSNDSDAYKDQISVLPNEQDLVREEAQKMSSLLPTMWLGAQNGCLYVHSSVAQWRKCLHSIKLKDSILSIVHVKGIVLVALADGTLAIFHRGVDGQWDLSNYHLLDLGRPHHSIRCMTVVHDKVWCGYRNKIYVVQPKAMKIEKSFDAHPRKESQVRQLAWVGDGVWVSIRLDSTLRLYHAHTYQHLQDVDIEPYVSKMLGTGKLGFSFVRITALMVSCNRLWVGTGNGVIISIPLTETNKTSGVPGNRPGSVIRVYGDENSDKVTPGTFIPYCSMAHAQLCFHGHRDAVKFFVAVPGQVISPQSSSSGTDLTGDKAGPSAQEPGSQTPLKSMLVISGGEGYIDFRMGDEGGESELLGEDLPLEPSVTKAERSHLIVWQVMYGNE  
sp|O60336|MABP1\_HUMAN Mitogen-activated protein kinase-binding protein 1 OS=Homo sapiens OX=9606 GN=MAPKBP1 PE=1 SV=4  
MAVEGSTITSRIKNLLRSPSIKLRRSKAGNRREDLSSKVTLEKVLGITVSGGRGLACDPRSGLVAYPAGCVVVLFNPRKHKQHHILNSSRKTITALAFSPDGKYLVTGESGHMPAVRVWDVAEHSQVAELQEHKYGVACVAFSPSAKYIVSVGYQHDMIVNVWAWKKNIVVASNKVSSRVTAVSFSEDCSYFVTAGNRHIKFWYLDDSKTSKVNATVPLLGRSGLLGELRNNLFTDVACGRGKKADSTFCITSSGLLCEFSDRRLLDKWVELRNIDSFTTTVAHCISVSQDYIFCGCADGTVRLFNPSNLHFLSTLPRPHALGTDIASVTEASRLFSGVANARYPDTIALTFDPTNQWLSCVYNDHSIYVWDVRDPKKVGKVYSALYHSSCVWSVEVYPEVKDSNQACLPPSSFITCSSDNTIRLWNTESSGVHGSTLHRNILSSDLIKIIYVDGNTQALLDTELPGGDKADASLLDPRVGIRSVCVSPNGQHLASGDRMGTLRVHELQSLSEMLKVEAHDSEILCLEYSKPDTGLKLLASASRDRLIHVLDAGREYSLQQTLDEHSSSITAVKFAASDGQVRMISCGADKSIYFRTAQKSGDGVQFTRTHHVVRKTTLYDMDVEPSWKYTAIGCQDRNIRIFNISSGKQKKLFKGSQGEDGTLIKVQTDPSGIYIATSCSDKNLSIFDFSSGECVATMFGHSEIVTGMKFSNDCKHLISVSGDSCIFVWRLSSEMTISMRQRLAELRQRQRGGKQQGPSSPQRASGPNRHQAPSMLSPGPALSSDSDKEGEDEGTEEELPALPVLAKSTKKALASVPSPALPRSLSHWEMSRAQESVGFLDPAPAANPGPRRRGRWVQPGVELSVRSMLDLRQLETLAPSLQDPSQDSLAIIPSGPRKHGQEALETSLTSQNEKPPRPQASQPCSYPHIIRLLSQEEGVFAQDLEPAPIEDGIVYPEPSDNPTMDTSEFQVQAPARGTLGRVYPGSRSSEKHSPDSACSVDYSSSCLSSPEHPTEDSESTEPLSVDGISSDLEEPAEGDEEEEEEEGGMGPYGLQEGSPQTPDQEQFLKQHFETLASGAAPGAPVQVPERSESRSISSRFLLQVQTRPLREPSPSSSSLALMSRPAQVPQASGEQPRGNGANPPGAPPEVEPSSGNPSPQQAASVLLPRCRLNPDSSWAPKRVATASPFSGLQKAQSVHSLVPQERHEASLQAPSPGALLSREIEAQDGLGSLPPADGRPSRPHSYQNPTTSSMAKISRSISVGENLGLVAEPQAHAPIRVSPLSKLALPSRAHLVLDIPKPLPDRPTLAAFSPVTKGRAPGEAEKPGFPVGLGKAHSTTERWACLGEGTTPKPRTECQAHPGPSSPCAQQLPVSSLFQGPENLQPPPPEKTPNPMECTKPGAALSQDSEPAVSLEQCEQLVAELRGSVRQAVRLYHSVAGCKMPSAEQSRIAQLLRDTFSSVRQELEAVAGAVLSSPGSSPGAVGAEQTQALLEQYSELLLRAVERRMERKL

### 15.3.1 Writing sequences

The write function takes three arguments — 1) a sequence object, 2) filename, and 3) file format. The code below reads a fasta file with multiple sequences and then save the first 10 sequences in a new file.

all\_seqs = []  
for seq\_record in SeqIO.parse("Human\_MAPK.fasta", "fasta"):  
 all\_seqs.append(seq\_record)  
print(len(all\_seqs))  
  
SeqIO.write(all\_seqs[0:5],"first\_5.fasta","fasta")

27

5

## 15.4 Multiple Sequence Alignment

The AlignIO class has functions to parse alignment files. The read and write functions have a similar syntax to the corresponding functions in the SeqIO class. The alignment object stores sequences in 2D array format such that the rows are number of sequences and columns represent alignment length. To extract a sub-set of an alignment, slicing feature can be used. The code below shows reading an alignment file in fasta format followed by selecting a portion of this alignment and save it in a new file in clustal format. The subset is extracted by giving the range for the rows and columns within square brackets. The numbering for both rows and columns starts from zero. In the example below first ten sequences in the alignment are selected since range of rows is :10 and the colums range is 3:12. The sequence file used in the code below is available [here](https://doi.org/10.1371/journal.pcbi.1005975.s002).

from Bio import AlignIO  
align1 = AlignIO.read("kinases.txt", "fasta")  
  
# slicing - [row range, col range]  
x = align1[:10,3:12]  
print(x)  
AlignIO.write(x,"msa1.aln","clustal")

Alignment with 10 rows and 9 columns  
LKVL-GKGA consensus  
IKEL-GRGN sp|Q22RR1||agc:agc-sar|Tetrahymena  
IKKL-GFGQ sp|Q234E6||agc:agc-sar|Tetrahymena  
VKKL-GNGQ sp|Q23KG5||agc:agc-sar|Tetrahymena  
IKTL-AFGQ sp|Q23DN8||agc:agc-sar|Tetrahymena  
IKKL-GVGQ sp|I7MFS4||agc:agc-sar|Tetrahymena  
IKKL-GFGQ sp|I7M3B5||agc:agc-sar|Tetrahymena  
IKKL-GFGQ sp|I7MD55||agc:agc-sar|Tetrahymena  
IKKL-GEGQ sp|Q869J9|pkg-2|agc:pkg|Paramecium  
IRVL-GKGC sp|A8N3F0||agc:agc-unique|Coprinopsis

1

# %load msa1.aln  
CLUSTAL X (1.81) multiple sequence alignment  
  
  
consensus LKVL-GKGA  
sp|Q22RR1||agc:agc-sar|Tetrahy IKEL-GRGN  
sp|Q234E6||agc:agc-sar|Tetrahy IKKL-GFGQ  
sp|Q23KG5||agc:agc-sar|Tetrahy VKKL-GNGQ  
sp|Q23DN8||agc:agc-sar|Tetrahy IKTL-AFGQ  
sp|I7MFS4||agc:agc-sar|Tetrahy IKKL-GVGQ  
sp|I7M3B5||agc:agc-sar|Tetrahy IKKL-GFGQ  
sp|I7MD55||agc:agc-sar|Tetrahy IKKL-GFGQ  
sp|Q869J9|pkg-2|agc:pkg|Parame IKKL-GEGQ  
sp|A8N3F0||agc:agc-unique|Copr IRVL-GKGC

## 15.5 Accessing databases using API

from Bio import Entrez, SeqIO  
Entrez.email = "you@example.com"  
handle1 = Entrez.efetch(db="nucleotide", id="NM\_001301717", rettype="gb", retmode="text")  
record1 = SeqIO.read(handle1, "genbank")

print(record1.description)  
print(len(record1.seq))  
print(record1.seq[:50])

Homo sapiens C-C motif chemokine receptor 7 (CCR7), transcript variant 4, mRNA  
2191  
CTCTAGATGAGTCAGTGGAGGGCGGGTGGAGCGTTGAACCGTGAAGAGTG

To retrieve multiple sequence, pass a comma-separated ids to the id keyword parameter. This time we’ll need to use the parse function to get the sequences from the efetch output.

all\_ids = ["NP\_000240.1", "NP\_000549.1", "NP\_001289656.1"]  
handle2 = Entrez.efetch(db="protein", id=",".join(all\_ids), rettype="gb", retmode="text")  
record2 = list(SeqIO.parse(handle2, "genbank"))

print(f"There are {len(record2)} sequences")  
for record in record2:  
 print(record.description)  
 print(len(record.seq))

There are 3 sequences  
DNA mismatch repair protein Mlh1 isoform 1 [Homo sapiens]  
756  
hemoglobin subunit alpha [Homo sapiens]  
142  
floral homeotic protein GLOBOSA-like [Eucalyptus grandis]  
208

## 15.6 Accessing databases without API

import requests  
  
uniprot\_id = "P12345" # Replace with the protein UniProt ID  
url = f"https://www.uniprot.org/uniprot/{uniprot\_id}.fasta"  
  
response = requests.get(url)  
print(response.text)

>sp|P12345|AATM\_RABIT Aspartate aminotransferase, mitochondrial OS=Oryctolagus cuniculus OX=9986 GN=GOT2 PE=1 SV=2  
MALLHSARVLSGVASAFHPGLAAAASARASSWWAHVEMGPPDPILGVTEAYKRDTNSKKM  
NLGVGAYRDDNGKPYVLPSVRKAEAQIAAKGLDKEYLPIGGLAEFCRASAELALGENSEV  
VKSGRFVTVQTISGTGALRIGASFLQRFFKFSRDVFLPKPSWGNHTPIFRDAGMQLQSYR  
YYDPKTCGFDFTGALEDISKIPEQSVLLLHACAHNPTGVDPRPEQWKEIATVVKKRNLFA  
FFDMAYQGFASGDGDKDAWAVRHFIEQGINVCLCQSYAKNMGLYGERVGAFTVICKDADE  
AKRVESQLKILIRPMYSNPPIHGARIASTILTSPDLRKQWLQEVKGMADRIIGMRTQLVS  
NLKKEGSTHSWQHITDQIGMFCFTGLKPEQVERLTKEFSIYMTKDGRISVAGVTSGNVGY  
LAHAIHQVTK

To get multiple sequences from Uniprot use the bulk query format

uniprot\_ids = ["P69905", "P68871"]   
  
query = " OR ".join([f"accession:{uid}" for uid in uniprot\_ids])  
url = f"https://rest.uniprot.org/uniprotkb/stream?format=fasta&query={query}"  
  
response = requests.get(url)  
response.raise\_for\_status() # Raise an error for failed requests

from io import StringIO  
fasta\_io = StringIO(response.text)  
records = list(SeqIO.parse(fasta\_io, "fasta"))

for record in records:  
 print(record.description)  
 print(len(record.seq))

sp|P68871|HBB\_HUMAN Hemoglobin subunit beta OS=Homo sapiens OX=9606 GN=HBB PE=1 SV=2  
147  
sp|P69905|HBA\_HUMAN Hemoglobin subunit alpha OS=Homo sapiens OX=9606 GN=HBA1 PE=1 SV=2  
142

## 15.7 Running BLAST over the internet

Biopython offers a functionality to programmatically run BLAST on the NCBI servers using the Bio.Blast class. To run blast online at NCBI servers, Bio.Blast can be used which has different function to run Blast and also to parse the output. The NCBIWWW library has qblast function that takes three arguments &emdash; 1) blast program (blastp, blastn, etc.), 2) database (any of the databases available at NCBI, and 3) sequence. Once the blast serach is over the output can be saved in a file. This output would be in XML format. We can use the read function within the NCBIXML class to parse this output. The code below shows running a blast search using qblast against the non-redundant database available at in NCBI. The output file saved in the previous step has all the hits identified in the Blast search. These hits follow a hierarchical manner such that each result would have multiple alignments and within each alignment would be multiple high scoring pairs (hsps) i.e. Blast object Alignment hsps. For more details on this you may refer to the Blast documentation available at NCBI.

from Bio.Blast import NCBIWWW  
from Bio.Blast import NCBIXML  
  
result\_ncbi = NCBIWWW.qblast("blastn", "nt", "8332116")  
  
with open("my\_blast.xml", "w") as file\_handle:  
 file\_handle.write(result\_ncbi.read())

result\_handle = open("my\_blast.xml")  
blast\_record = NCBIXML.read(result\_handle)  
count = 0  
for alignment in blast\_record.alignments:  
 for hsp in alignment.hsps:  
 print(hsp)  
 count += 1  
 if(count==5):  
 break

Score 482 (435 bits), expectation 5.6e-117, alignment length 624  
Query: 59 ACAGAAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGA...GTA 678  
 || ||||||||| |||| | |||| || |||| |||| | ||||... ||  
Sbjct: 278 ACCGAAAATGGGCAGAGGAGTGAATTATATGGCAATGACACCTGA...TTA 901  
Score 463 (418 bits), expectation 1.5e-111, alignment length 590  
Query: 63 AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAA...CCC 649  
 |||||||| ||| |||| | || ||||| |||||||| || |||...|||  
Sbjct: 42 AAAATGGGTAGACGAATGGATTATTTGGCGATGAAAACCGAGCAA...CCC 631  
Score 443 (400 bits), expectation 4.0e-106, alignment length 597  
Query: 87 TTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGAT...TAG 679  
 ||||||||||||||||| ||| |||| |||||||| |||| |||...|||  
Sbjct: 25 TTGGCCATGAAAACTGAGCAAATGGCGTTGGCTAATTTGATAGAT...TAG 621  
Score 441 (398 bits), expectation 1.4e-105, alignment length 593  
Query: 65 AATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATT...CTG 655  
 ||||||||| ||| | | | ||||||||||||||||||| ... ||  
Sbjct: 254 AATGGGGAG-GAA--GGATAATTTGGCCATGAAAACTGATCC---...ATG 838  
Score 439 (397 bits), expectation 4.9e-105, alignment length 596  
Query: 63 AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAA...CTG 655  
 ||||||||||| ||| ||| ||||| |||| |||||||| |...|||  
Sbjct: 169 AAAATGGGGAGG---ATGGAGTTTTTGGCTATGAGAACTGATCCA...CTG 755

The hsps object has several attributes including the BLAST statistics such as evalue, score, positives, etc. These can be used to extract hits based on certain conditions. E.g., the code below shows saving hits from the previous Blast search with evalue greater than 1e-105 to a new file.

with open("new\_file.txt", "w") as file\_handle:  
 for alignment in blast\_record.alignments:  
 for hsp in alignment.hsps:  
 if (hsp.expect < 1e-105):  
 print(hsp)  
 file\_handle.write(str(hsp)+"\n")  
 file\_handle.write("\n")  
print("DONE")

Score 482 (435 bits), expectation 5.6e-117, alignment length 624  
Query: 59 ACAGAAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGA...GTA 678  
 || ||||||||| |||| | |||| || |||| |||| | ||||... ||  
Sbjct: 278 ACCGAAAATGGGCAGAGGAGTGAATTATATGGCAATGACACCTGA...TTA 901  
Score 463 (418 bits), expectation 1.5e-111, alignment length 590  
Query: 63 AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAA...CCC 649  
 |||||||| ||| |||| | || ||||| |||||||| || |||...|||  
Sbjct: 42 AAAATGGGTAGACGAATGGATTATTTGGCGATGAAAACCGAGCAA...CCC 631  
Score 443 (400 bits), expectation 4.0e-106, alignment length 597  
Query: 87 TTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGAT...TAG 679  
 ||||||||||||||||| ||| |||| |||||||| |||| |||...|||  
Sbjct: 25 TTGGCCATGAAAACTGAGCAAATGGCGTTGGCTAATTTGATAGAT...TAG 621  
DONE

## 15.8 Creating a dataframe of BLAST results

The better approach to work with BLAST results is to create a dataframe having all the attributes for all the hits. The following code iterates through the BLAST output and creates a pandas dataframe having all the results.

import pandas as pd  
  
result\_handle = open("my\_blast.xml")  
blast\_record = NCBIXML.read(result\_handle)  
blast\_data = []  
  
for alignment in blast\_record.alignments:  
 for hsp in alignment.hsps:  
 # append a dictionary of attributes for each hit  
 blast\_data.append({  
 "query\_id": blast\_record.query\_id,  
 "hit\_id": alignment.hit\_id,  
 "hit\_def": alignment.hit\_def,  
 "accession": alignment.accession,  
 "hit\_length": alignment.length,  
  
 "e\_value": hsp.expect,  
 "bit\_score": hsp.bits,  
 "score": hsp.score,  
 "alignment\_length": hsp.align\_length,  
 "identity": hsp.identities,  
 "positives": hsp.positives,  
 "gaps": hsp.gaps,  
 "percent\_identity": 100 \* hsp.identities / hsp.align\_length,  
 "percent\_positives": 100 \* hsp.positives / hsp.align\_length,  
  
 "query\_start": hsp.query\_start,  
 "query\_end": hsp.query\_end,  
 "subject\_start": hsp.sbjct\_start,  
 "subject\_end": hsp.sbjct\_end,  
   
 "query\_sequence": hsp.query,  
 "match\_sequence": hsp.match,  
 "subject\_sequence": hsp.sbjct,  
 })  
  
# Create DataFrame  
df = pd.DataFrame(blast\_data)  
  
print(df.shape)  
df.head()

(50, 23)

|  | query\_id | hit\_id | hit\_def | accession | hit\_length | e\_value | bit\_score | score | alignment\_length | identity | ... | percent\_positives | query\_start | query\_end | subject\_start | subject\_end | query\_frame | strand | query\_sequence | match\_sequence | subject\_sequence |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | BE037100.1 | gi|1219041180|ref|XM\_021875076.1| | PREDICTED: Chenopodium quinoa cold-regulated 4... | XM\_021875076 | 1173 | 5.569490e-117 | 435.898 | 482.0 | 624 | 473 | ... | 75.801282 | 59 | 678 | 278 | 901 | None | (Plus, Plus) | ACAGAAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGAT... | || ||||||||| |||| | |||| || |||| |||| | |||| ... | ACCGAAAATGGGCAGAGGAGTGAATTATATGGCAATGACACCTGAG... |
| 1 | BE037100.1 | gi|2514617377|ref|XM\_021992092.2| | PREDICTED: Spinacia oleracea cold-regulated 41... | XM\_021992092 | 752 | 1.494500e-111 | 418.766 | 463.0 | 590 | 447 | ... | 75.762712 | 63 | 649 | 42 | 631 | None | (Plus, Plus) | AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAAT... | |||||||| ||| |||| | || ||||| |||||||| || ||||... | AAAATGGGTAGACGAATGGATTATTTGGCGATGAAAACCGAGCAAT... |
| 2 | BE037100.1 | gi|2518612504|ref|XM\_010682658.3| | PREDICTED: Beta vulgaris subsp. vulgaris cold-... | XM\_010682658 | 621 | 4.010310e-106 | 400.732 | 443.0 | 597 | 448 | ... | 75.041876 | 87 | 679 | 25 | 621 | None | (Plus, Plus) | TTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGATT... | ||||||||||||||||| ||| |||| |||||||| |||| ||||... | TTGGCCATGAAAACTGAGCAAATGGCGTTGGCTAATTTGATAGATT... |
| 3 | BE037100.1 | gi|2031543140|ref|XM\_041168865.1| | PREDICTED: Juglans microcarpa x Juglans regia ... | XM\_041168865 | 1020 | 1.399730e-105 | 398.929 | 441.0 | 593 | 448 | ... | 75.548061 | 65 | 655 | 254 | 838 | None | (Plus, Plus) | AATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATTG... | ||||||||| ||| | | | ||||||||||||||||||| |... | AATGGGGAG-GAA--GGATAATTTGGCCATGAAAACTGATCC---G... |
| 4 | BE037100.1 | gi|2618480339|ref|XM\_048479995.2| | PREDICTED: Ziziphus jujuba cold-regulated 413 ... | XM\_048479995 | 1028 | 4.885550e-105 | 397.126 | 439.0 | 596 | 448 | ... | 75.167785 | 63 | 655 | 169 | 755 | None | (Plus, Plus) | AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAAT... | ||||||||||| ||| ||| ||||| |||| |||||||| | ... | AAAATGGGGAGG---ATGGAGTTTTTGGCTATGAGAACTGATCCA-... |

df.columns

Index(['query\_id', 'hit\_id', 'hit\_def', 'accession', 'hit\_length', 'e\_value',  
 'bit\_score', 'score', 'alignment\_length', 'identity', 'positives',  
 'gaps', 'percent\_identity', 'percent\_positives', 'query\_start',  
 'query\_end', 'subject\_start', 'subject\_end', 'query\_frame', 'strand',  
 'query\_sequence', 'match\_sequence', 'subject\_sequence'],  
 dtype='object')

Now that we have a dataframe, we can easily select hits based on desired thresholds. For instance, the code below selects all the hits with evalue less than 1e-105.

df[df["e\_value"] < 1e-105]

|  | query\_id | hit\_id | hit\_def | e\_value | bit\_score | score | query\_start | query\_end | subject\_start | subject\_end | alignment\_length | identity | gaps | query\_sequence | match\_sequence | subject\_sequence |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | BE037100.1 | gi|1219041180|ref|XM\_021875076.1| | PREDICTED: Chenopodium quinoa cold-regulated 4... | 5.569490e-117 | 435.898 | 482.0 | 59 | 678 | 278 | 901 | 624 | 473 | 4 | ACAGAAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGAT... | || ||||||||| |||| | |||| || |||| |||| | |||| ... | ACCGAAAATGGGCAGAGGAGTGAATTATATGGCAATGACACCTGAG... |
| 1 | BE037100.1 | gi|2514617377|ref|XM\_021992092.2| | PREDICTED: Spinacia oleracea cold-regulated 41... | 1.494500e-111 | 418.766 | 463.0 | 63 | 649 | 42 | 631 | 590 | 447 | 3 | AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAAT... | |||||||| ||| |||| | || ||||| |||||||| || ||||... | AAAATGGGTAGACGAATGGATTATTTGGCGATGAAAACCGAGCAAT... |
| 2 | BE037100.1 | gi|2518612504|ref|XM\_010682658.3| | PREDICTED: Beta vulgaris subsp. vulgaris cold-... | 4.010310e-106 | 400.732 | 443.0 | 87 | 679 | 25 | 621 | 597 | 448 | 4 | TTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGATT... | ||||||||||||||||| ||| |||| |||||||| |||| ||||... | TTGGCCATGAAAACTGAGCAAATGGCGTTGGCTAATTTGATAGATT... |

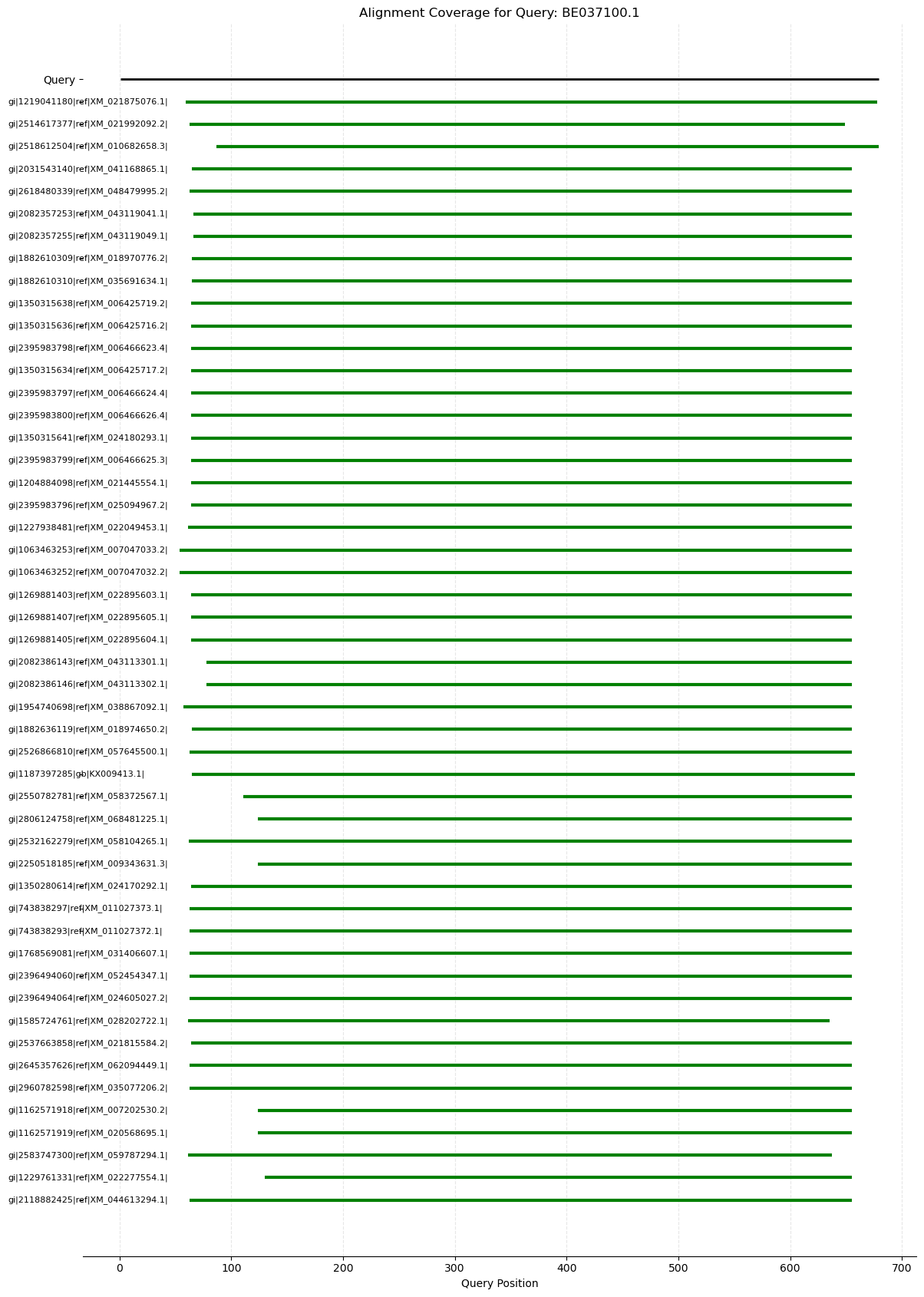
To save the filtered set of hits (or the entire output) to a csv file, we can use the to\_csv function.

df\_top\_hits = df[df["e\_value"] < 1e-105]  
df\_top\_hits.to\_csv("Blast\_hits.csv", index=False)

# %load Blast\_hits.csv  
query\_id,hit\_id,hit\_def,e\_value,bit\_score,score,query\_start,query\_end,subject\_start,subject\_end,alignment\_length,identity,gaps,query\_sequence,match\_sequence,subject\_sequence  
BE037100.1,gi|1219041180|ref|XM\_021875076.1|,"PREDICTED: Chenopodium quinoa cold-regulated 413 plasma membrane protein 2-like (LOC110697660), mRNA",5.56949e-117,435.898,482.0,59,678,278,901,624,473,4,ACAGAAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGATTCCGATATCAATGAGCTTAAAATGGCAACAATGAGGCTCATCAATGATGCTAGTATGCTCGGT--CATTACGGG-TTTGGCACTCATTTCCTCAAATGGCTCGCCTGCCTTGCGGCTATTTACTTGTTGATATTGGATCGAACAAACTGGAGAACCAACATGCTCACGTCACTTTTAGTCCCTTACATATTCCTCAGTCTTCCATCCGGGCCATTTCATCTGTTCAGAGGCGAGGTCGGGAAATGGATTGCCATCATTGCAGTCGTGTTAAGGCTGTTCTTCAACCGGCATTTCCCAGTTTGGCTGGAAATGCCTGGATCGTTGATACTCCTCCTGGTGGTGGCACCAGACTTCTTTACACACAAAGTGAAGGAGAGCTGGATCGGAATTGCAATTATGATAGCGATAGGGTGTCACCTGATGCAAGAACATATCAGAGCCACTGGTGGCTTTTGGAATTCCTTCACACAGAGCCACGGAACTTTTAACACAATTGGGCTTATCCTTCTACTGGCTTACCCTGTCT-GTTTATGGTCATCTTCATGATGTA,|| ||||||||| |||| | |||| || |||| |||| | |||| ||| | |||| ||| ||| ||||| | ||||| ||||||||||| || || | |||| ||||| |||||||| || ||||| || | || ||||| |||||||| || | |||||| |||| ||||| ||||||||||||||||| ||||||| ||||| |||||||||||||||||||| ||||| ||||| || |||||||||||||||||||| || || || ||| | || | || || ||||| |||||||||||||| | | || || ||| ||||||| |||||| |||| | ||||||| ||| | || ||||| ||||| ||||||| | ||||||| || ||||| || | |||||||| | ||||| || ||||| ||| ||| |||| || || ||||||||| | || |||||||||||||||| || |||||||| || | ||||| || || || ||||| |||| || | || |||||||| | || || |||||||||| | | || ||||| || |||||||| ||,ACCGAAAATGGGCAGAGGAGTGAATTATATGGCAATGACACCTGAGCAACTAGCCGCGGCCAATTTGATCAACTCCGACATCAATGAGCTCAAGATCGTTGTGATGACACTCATTCATGATGCTTCTAGACTCGGCGGCACCTCAGGATTTGGAACTCATTTTCTTAGATGGCTAGCCTCTCTTGCTGCTATTTACTTGTTGATCCTGGATCGCACAAATTGGAGAACCAACATGCTCACATCACTCTTAGTACCATACATATTCCTCAGTCTTCCTTCTGGCCCTTTTTACCTTCTTAGGGGTGAGGTTGGGAAATGGATTGCTTTTGTCGCGGTTGTGCTAAGGCTATTCTTCCACCGCCGCTTCCCAGAATGGTTAGAGATGCCAGGATCACTGATACTATTGTTGGTGGTAGCTCCAGAATTGCTAGCACACAAATTAAAGGATAGTTGGATGGGAGTTGTAATTCTGTTAATCATAGGGTGTTATTTGCTGCAAGAACATATCAGGGCAACTGGTGGTTTAAGAAATTCGTTTACTCAAAGCCATGGAATTTCCTATACGATTGGGCTGCTTCTCTTATTGGCTTACCCAATTTGGTCCATGGTTATTTTCATGATTTA  
BE037100.1,gi|2514617377|ref|XM\_021992092.2|,"PREDICTED: Spinacia oleracea cold-regulated 413 plasma membrane protein 2-like (LOC110787470), mRNA",1.4945e-111,418.766,463.0,63,649,42,631,590,447,3,AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGATTCCGATATCAATGAGCTTAAAATGGCAACAATGAGGCTCATCAATGATGCTAGTATGCTCGG---TCATTACGGGTTTGGCACTCATTTCCTCAAATGGCTCGCCTGCCTTGCGGCTATTTACTTGTTGATATTGGATCGAACAAACTGGAGAACCAACATGCTCACGTCACTTTTAGTCCCTTACATATTCCTCAGTCTTCCATCCGGGCCATTTCATCTGTTCAGAGGCGAGGTCGGGAAATGGATTGCCATCATTGCAGTCGTGTTAAGGCTGTTCTTCAACCGGCATTTCCCAGTTTGGCTGGAAATGCCTGGATCGTTGATACTCCTCCTGGTGGTGGCACCAGACTTCTTTACACACAAAGTGAAGGAGAGCTGGATCGGAATTGCAATTATGATAGCGATAGGGTGTCACCTGATGCAAGAACATATCAGAGCCACTGGTGGCTTTTGGAATTCCTTCACACAGAGCCACGGAACTTTTAACACAATTGGGCTTATCCTTCTACTGGCTTACCC,|||||||| ||| |||| | || ||||| |||||||| || ||||| |||| ||| ||| ||||||||||||||||||| ||||| || || || ||| |||| | ||||| ||| || |||||| ||| | || || || |||||||| ||| |||||||||||| || | |||||||||||||| || | ||||||||| | ||||||| ||||||||| || ||||||||||| || ||||| ||||| ||||| ||||| || || ||||| || || ||||| ||||| |||||||||||||| | | | || ||| ||||| | |||||| |||| |||||||| ||| |||||||||| | || |||||||| | ||||||| || ||||| | || || | |||| ||||| ||||| ||| ||| | || || || ||||| ||| |||| | ||||| ||||| || || |||||||| || ||||||| |||||||| || || || | || ||||||||||||||| | || | ||||||||||,AAAATGGGTAGACGAATGGATTATTTGGCGATGAAAACCGAGCAATTAGCCGCGGCCAATTTGATCGATTCCGATATCAACGAGCTGAAGATCGCCGTGATGGCGCTCGTTCATGATACTACTACGCTCGGCGGTCAATCGGGATTCGGAACTCATTTTCTCCAATGGCTCGCCTCATTTTCTGCTATTTACTTGTTAATCCTTGATCGAACACATTGGAGAAGCAACATGCTTACTTCACTTTTAGTACCATACATTTTCCTAAGTCTCCCATCTGGCCCCTTTCACCTTTTAAGAGGTGAGGTTGGGAAATGGATTGCTTTTGTCTCGGTTGTGCTAAGGTTATTCTTCCACCGCAGTTTCCCAGAATGGTTGGAAATGCCAGTATGTTTGATACTATTATTGGTGGTAGCTCCAGAAATGCTTGCAATATCAATGAAAGAGAGTTGGATGGGAGTTGTAGTTGTGTTAATCATAGGATGTTACCTTCTACAAGAGCATATTAGGGCAACTGGTGGTTTAAGGAATTCTTTCACACAAAGACATGGGATTTCCAACACAATTGGGCTTCTTCTCTTGTTGGCTTACCC  
BE037100.1,gi|2518612504|ref|XM\_010682658.3|,"PREDICTED: Beta vulgaris subsp. vulgaris cold-regulated 413 plasma membrane protein 2 (LOC104895996), mRNA",4.01031e-106,400.732,443.0,87,679,25,621,597,448,4,TTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGATTCCGATATCAATGAGCTTAAAATGGCAACAATGAGGCTCATCAATGATGCTAGTATGCTC---GGTCATTACGGGTTTGGCACTCATTTCCTCAAATGGCTCGCCTGCCTTGCGGCTATTTACTTGTTGATATTGGATCGAACAAACTGGAGAACCAACATGCTCACGTCACTTTTAGTCCCTTACATATTCCTCAGTCTTCCATCCGGGCCATTTCATCTGTTCAGAGGCGAGGTCGGGAAATGGATTGCCATCATTGCAGTCGTGTTAAGGCTGTTCTTCAACCGGCATTTCCCAGTTTGGCTGGAAATGCCTGGATCGTTGATACTCCTCCTGGTGGTGGCACCAGACTTCTTTACACACAAAGTGAAGGAGAGCTGGATCGGAATTGCAATTATGATAGCGATAGGGTGTCACCTGATGCAAGAACATATCAGAGCCACTGGTGGCTTTTGGAATTCCTTCACACAGAGCCACGGAACTTTTAACACAATTGGGCTTATCCTTCTACTGGCTTACCCTGTCT-GTTTATGGTCATCTTCATGATGTAG,||||||||||||||||| ||| |||| |||||||| |||| |||| ||||| ||||| ||||| || || | ||| || |||||||| || ||| ||||| || || || |||||||| ||||||||||| || | |||| |||||||| ||| |||| ||||||||||||| |||||||| |||||||| || |||||||||||||||||||||||| |||| ||||| | || ||||| || || || ||||| || || || |||||||| | || ||||| || | ||| | || ||| |||| |||||||||| |||| |||||||||| |||| |||||||| | ||||||| || || | || ||| | | | | ||||||| |||| | || ||| | || || || ||||||||| | || | ||||| ||||| ||||| || ||||| | ||||||||||||| || | ||| || | || ||| |||||||||||| |||||||| |||||||||| | | || || |||| ||||||||||||,TTGGCCATGAAAACTGAGCAAATGGCGTTGGCTAATTTGATAGATTATGATATGAATGAACTTAAGATCGCTTTGACATCGCTAATTCATGATGCTGCTAGACTCAATGGTCACGGAGGATTCGGAACTCATTTTCTCAAATGGCTTGCTTCATTTGCTGCTATTTATTTGCTGATCCTGGATCGAACAAATTGGAGAACAAACATGCTAACATCACTTTTAGTCCCTTACATATTCTTCAGCCTTCCCACAGGCATCTTTCACCTTTTAAGGGGCGAAGTTGGTAAGTGGATTGCTTTTATCGCAGTTGTACTGAGGTTATTTTTCCACCGCCATTTCCCAGATTGGTTGGAAATGCCAGGATGTTTGATACTGTTGTTGGTGGTCGCTCCCAATTTATTTGCTTATACAATGAAGGATGACTGGTTGGGGGTTGTAGTTTTGTTAATCATAGGGTGTTATTTGCTACAAGAGCATATAAGAGCGACGGGTGGTCTCAGGAATTCCTTCACTCAAACCCATGGTATTTCTAATACAATTGGGCTTCTCCTTCTATTGGCTTACCCCATTTGGTCGATATTCATATTCATGATGTAG

## 15.9 Plotting BLAST result

import matplotlib.pyplot as plt  
  
# Define bins and corresponding NCBI BLAST-style colors  
bins = [0, 50, 70, 80, 90, 100, float('inf')]  
colors = ['blue', 'cyan', 'green', 'darkorange', 'magenta', 'red']  
  
# Assign color based on percent identity  
df['color'] = pd.cut(df['percent\_identity'], bins=bins, labels=colors, right=True)  
  
# Plot per query  
for query in df['query\_id'].unique():  
 sub\_df = df[df['query\_id'] == query]  
  
 plt.figure(figsize=(12, 2 + len(sub\_df) \* 0.3))  
 query\_length = sub\_df['query\_end'].max()  
 plt.hlines(y=0, xmin=1, xmax=query\_length, colors='black', linewidth=2, label='Query')  
  
 for i, row in enumerate(sub\_df.itertuples(), start=1):  
 plt.hlines(y=-i, xmin=row.query\_start, xmax=row.query\_end, colors=row.color, linewidth=3)  
 plt.text(-100, -i, f"{row.hit\_id}", va='center', fontsize=8)  
  
 plt.title(f"Alignment Coverage for Query: {query}")  
 plt.xlabel("Query Position")  
 plt.yticks([0] + [-i for i in range(1, len(sub\_df)+1)], ['Query'] + ['']\*len(sub\_df))  
 plt.grid(True, axis='x', linestyle='--', alpha=0.3)  
 plt.gca().spines[['top', 'right', 'left']].set\_visible(False)  
 plt.tight\_layout()  
 plt.show()



## 15.10 BLAST search using sequence file

To run the Blast search using a sequence file instead of gi number, we first need to create a seqeunce object and then pass it on to the qblast function as shown below. To run this code, save the protein sequence below in a new file example1.fasta.

MFHPGMTSQPSTSNQMYYDPLYGAEQIVQCNPMDYHQANILCGMQYFNNSHNRYPLLPQMPPQFTNDHPY DFPNVPTISTLDEASSFNGFLIPSQPSSYNNNNISCVFTPTPCTSSQASSQPPPTPTVNPTPIPPNAGAV LTTAMDSCQQISHVLQCYQQGGEDSDFVRKAIESLVKKLKDKRIELDALITAVTSNGKQPTGCVTIQRSL DGRLQVAGRKGVPHVVYARIWRWPKVSKNELVKLVQCQTSSDHPDNICINPYHYERVVSNRITSADQSLH VENSPMKSEYLGDAGVIDSCSDWPNTPPDNNFNGGFAPDQPQLVTPIISDIPIDLNQIYVPTPPQLLDNW CSIIYYELDTPIGETFKVSARDHGKVIVDGGMDPHGENEGRLCLGALSNVHRTEASEKARIHIGRGVELT AHADGNISITSNCKIFVRSGYLDYTHGSEYSSKAHRFTPNESSFTVFDIRWAYMQMLRRSRSSNEAVRAQ AAAVAGYAPMSVMPAIMPDSGVDRMRRDFCTIAISFVKAWGDVYQRKTIKETPCWIEVTLHRPLQILDQL LKNSSQFGSS

seq\_file = open('example1.fasta')  
result\_handle2 = NCBIWWW.qblast("blastp", "nr", seq\_file.read())  
seq\_file.close()  
  
with open("test\_blast.xml", "w") as out\_handle:  
 out\_handle.write(result\_handle2.read())  
  
blast\_output = open("test\_blast.xml")   
blast\_record = NCBIXML.read(blast\_output)  
print(blast\_record.alignments[0])

sp|P45897.1| RecName: Full=Dwarfin sma-4; AltName: Full=MAD protein homolog 3 [Caenorhabditis elegans] >gb|AAA97605.1| SMA-4 [Caenorhabditis elegans]  
 Length = 570

Let’s say we need only the alignment with the mouse sequence, then, to print first 50 characters of each alignment with the mouse sequence along with corresponding statistics, the following code can be used.

for alignment in blast\_record.alignments:  
 if "Mus musculus" in alignment.title:  
 print(alignment.title)  
 for hsp in alignment.hsps:  
 print(hsp.query[0:50])  
 print(hsp.match[0:50])  
 print(hsp.sbjct[0:50])  
 print(hsp.positives, hsp.score, hsp.expect)

gb|KAI2586911.1| SMAD family member 4 [Homo sapiens] >gb|KAI4046350.1| SMAD family member 4 [Homo sapiens] >gb|PNI70022.1| SMAD4 isoform 6 [Pan troglodytes] >gb|PNJ43713.1| SMAD4 isoform 9 [Pongo abelii]  
TAMDSCQQISHVLQCYQQGGEDSDFVRKAIESLVKKLKDKRIELDALITA  
T+ D+C I H L C++QGGE F ++AIESLVKKLK+K+ ELD+LITA  
TSNDACLSIVHSLMCHRQGGESETFAKRAIESLVKKLKEKKDELDSLITA  
274 795.0 8.00678e-96