Python Programming

Release 1.0.0

Copyright & Disclaimer

B. TECH CSE with Specialization in Python Programming

Version 1.0.0

Copyright and Trademark Information for Partners/Stakeholders.

Copyright © 2018 Xebia. All rights reserved.

Please note that the information contained in this classroom material is subject to change without notice. Furthermore, this material contains proprietary information that is protected by copyright. No part of this material may be photocopied, reproduced, or translated to another language without the prior consent of Xebia or ODW Inc. Any such complaints can be raised at sales@odw.rocks

The language used in this course is US English. Our sources of reference for grammar, syntax, and mechanics are from The Chicago Manual of Style, The American Heritage Dictionary, and the Microsoft Manual of Style for Technical Publications.

Acknowledgements

We would like to sincerely thank the experts who have contributed to and shaped B. TECH CSE with Specialization in Data science and machine learning. Version 1.0.0 Semester 2

SME

Suresh K

He was in IT field for the past 26 years after completing his M.E. from IISc, Bangalore. AWS Solutions Architect – Professional and a consultant with specialization in areas of Cloud, Data Storage and Big Data. He worked as Cloud Architect in Oracle's Managed Cloud Services Group. He had also worked as General Manager in IBM and Group Head at Wipro in charge of the data science vertical.

Course Reviewers.

Veer Abhimanyu Singh | Xebia

Jaya Singh | Xebia

Sherry Wasandi | Xebia

Review Board Members.

Anand Sahay | Xebia

Kalyan Gali | ODW

Xebia Group consists of seven specialized, interlinked companies: Xebia, Xebia Academy, XebiaLabs, StackState, GoDataDriven, Xpirit and Binx.io. With offices in Amsterdam and Hilversum (Netherlands), Paris, Delhi, Bangalore and Boston, we employ over 700 people worldwide. Our solutions address digital strategy; agile transformations; DevOps and continuous delivery; big data and data science; cloud infrastructures; agile software development; quality and test automation; and agile software security.

ODW is dedicated to provide innovative and creative solutions that contribute in growth of emerging technologies. As a learning experience provider, ODW strengths include providing unique, up to date content by combining industry best practices with leading edge technology. ODW delivers high quality solutions and services which focus on digital learning transformation.

Python Programming Lab

Prerequisites

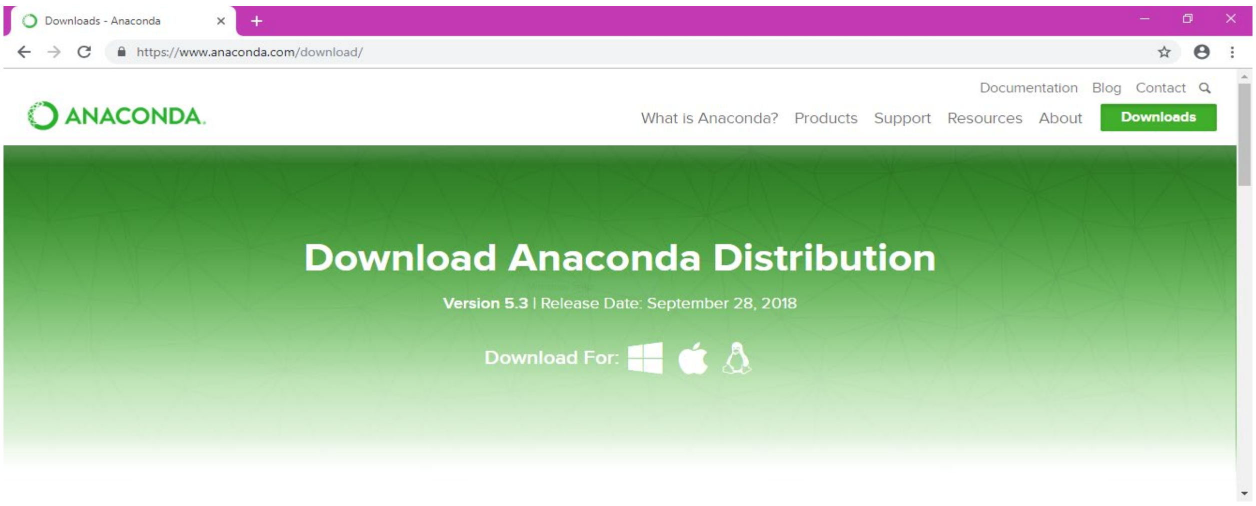
- 64-bit Windows OS is preferred
- Minimum 2 GB RAM with at least 5 GB Free Disk space is required.
- Python version 3.6.5 standard build for Windows / Linux
- IDEs such as VScode, Sublime or Atom is preferred
- Internet connectivity for downloading and installing Python using Anaconda

1.1. Python Environment using Anaconda

Anaconda is a open source package manager for managing python binaries, environments and their packages. There is a group of community maintaining and managing this very frequently. This process covers the installation of Anaconda and python in a 64-bit Windows 10 Operating system.

Steps:

1. Check if any python versions are already installed. If yes, uninstall them completely. Windows Add or Remove programs can be used to find any existing python installations.
2. Download the Anaconda installer from the following URL. https://repo.anaconda.com/archive/Anaconda3-5.3.0-Windows-x86_64.exe
   The installers for other platforms can be found from the following URL. https://www.anaconda.com/download/

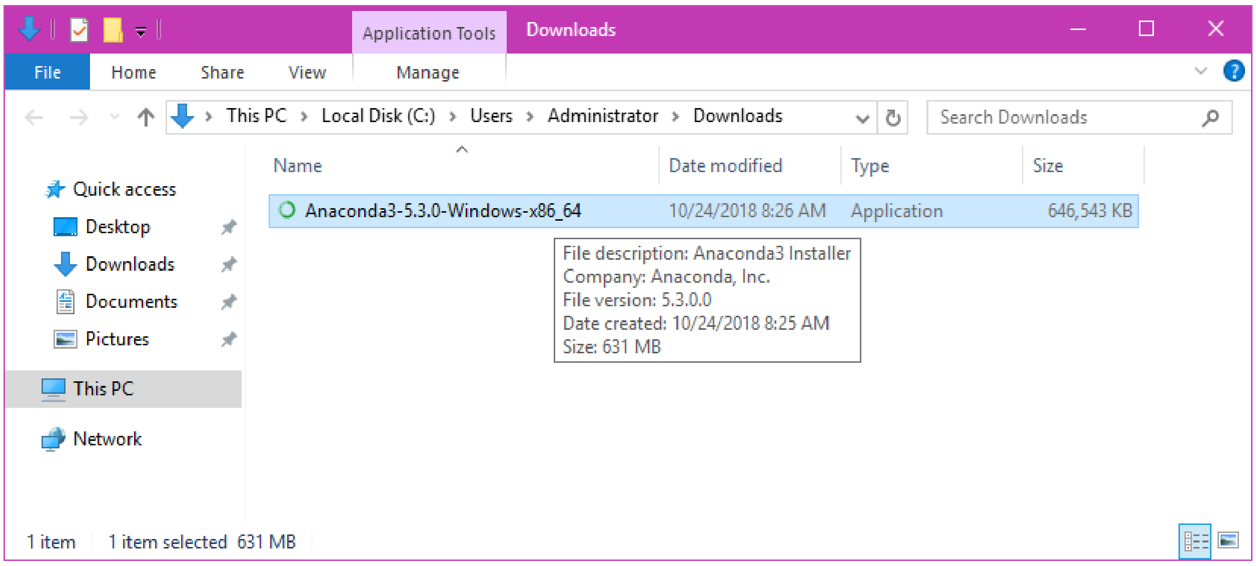


Download Anaconda Distribution

Version 5.3 | Release Date: September 28, 2018

Download For:

3. Open the downloaded executable file.

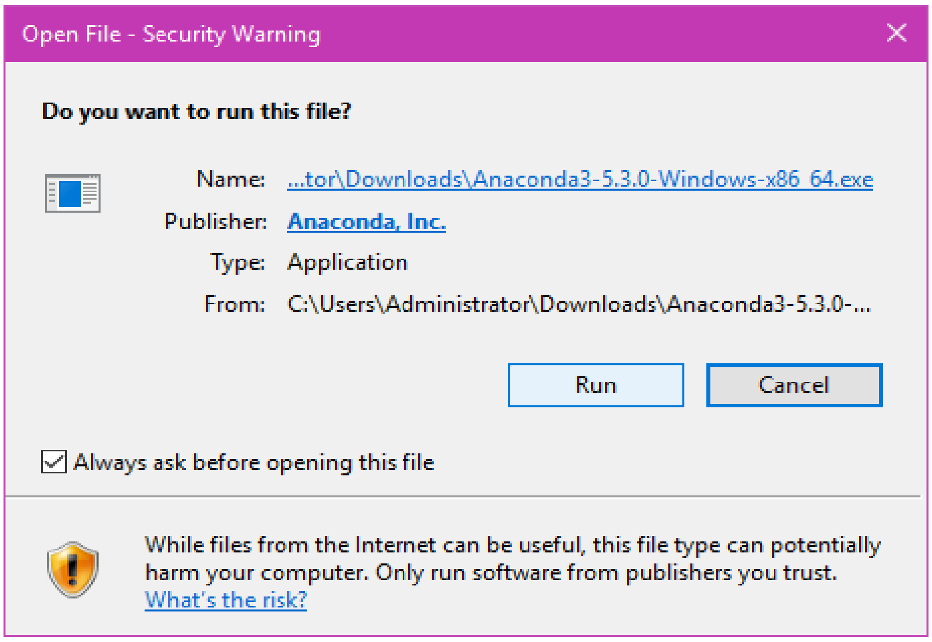


Name	Date modified	Type	Size
Anaconda3-5.3.0-Windows-x86_64	10/24/2018 8:26 AM	Application	646,543 KB

File description: Anaconda3 Installer
Company: Anaconda, Inc.
File version: 5.3.0.0
Date created: 10/24/2018 8:25 AM
Size: 631 MB

1 item 1 item selected 631 MB

4. Click Run if the windows security dialog prompts.


Open File - Security Warning

Do you want to run this file?

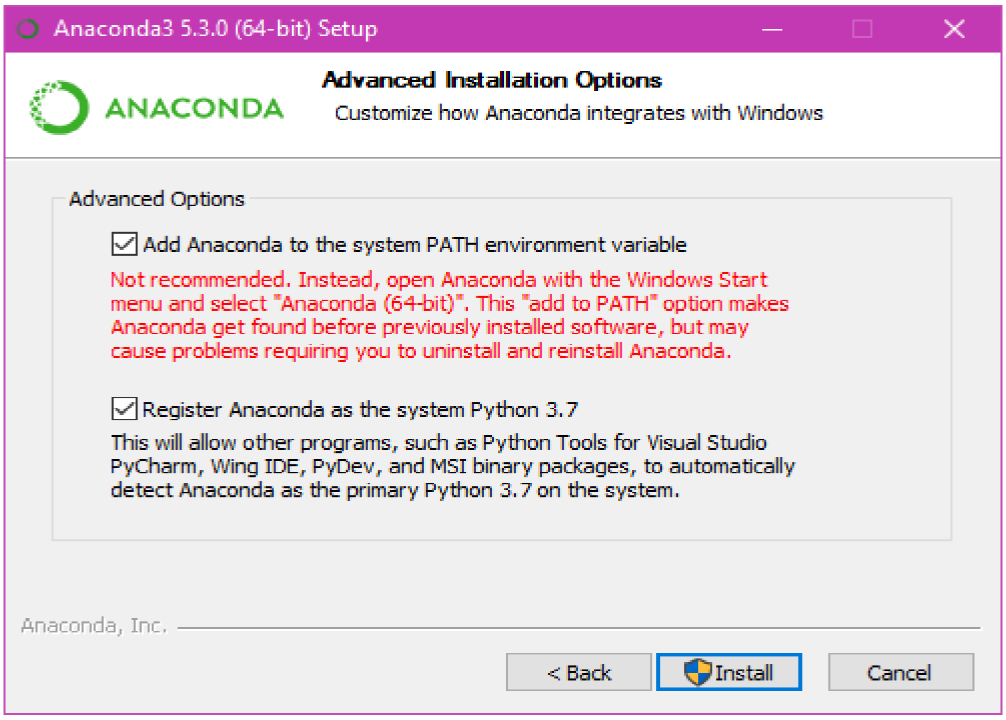
Name: ...tor\Downloads\Anaconda3-5.3.0-Windows-x86_64.exe
Publisher: Anaconda, Inc.
Type: Application
From: C:\Users\Administrator\Downloads\Anaconda3-5.3.0-...

☑ Always ask before opening this file

While files from the Internet can be useful, this file type can potentially harm your computer. Only run software from publishers you trust. What's the risk?

5. Click Next with the default values till the “Advanced Installation Options” page.

6. In "Advanced Installation Options" page, enable "Add anaconda to system PATH environment variable" and click the Install button.



Advanced Installation Options

Customize how Anaconda integrates with Windows

Advanced Options

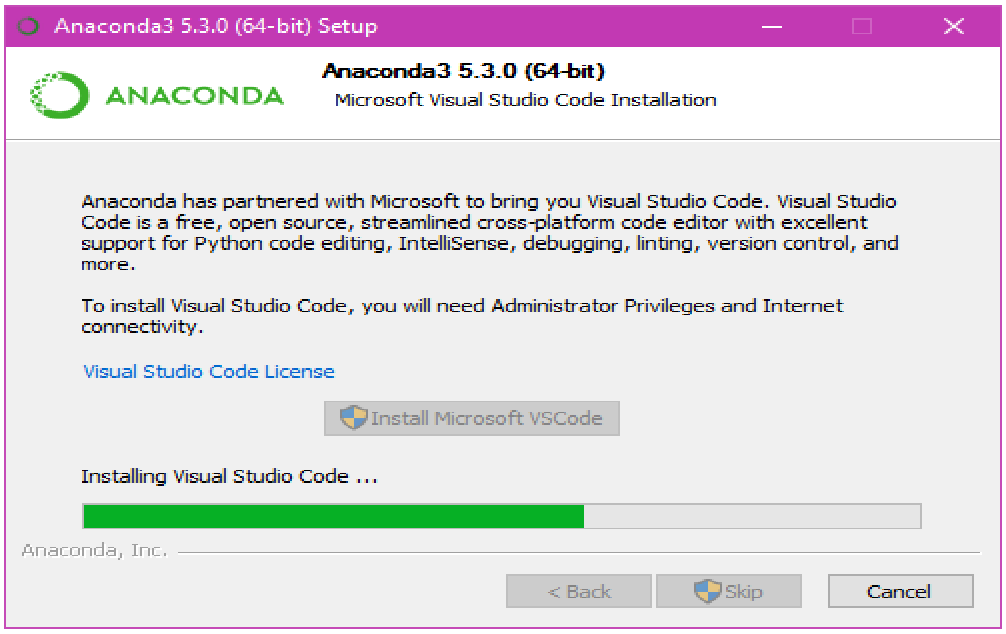
☑ Add Anaconda to the system PATH environment variable

Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

☑ Register Anaconda as the system Python 3.7

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.7 on the system.

7. It takes around 20 minutes for the installation to get completed. Once done, click Next.
8. It prompts to install Microsoft VScode. Install it if VScode is not already installed. This IDE will facilitate in writing python code and is not mandatory to run any python programs.



Anaconda3 5.3.0 (64-bit)

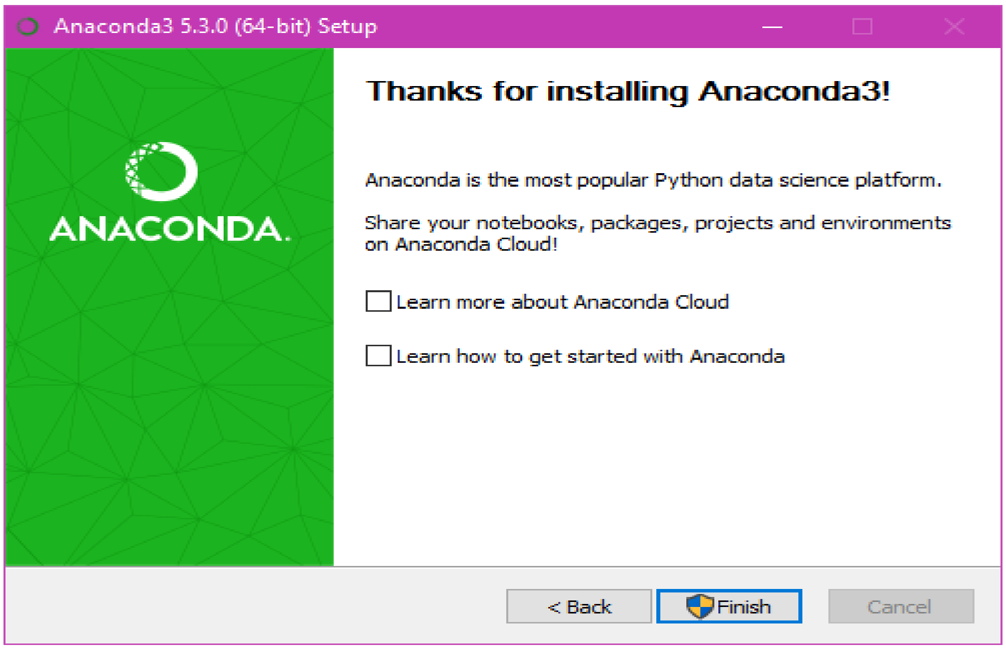
Microsoft Visual Studio Code Installation

Anaconda has partnered with Microsoft to bring you Visual Studio Code. Visual Studio Code is a free, open source, streamlined cross-platform code editor with excellent support for Python code editing, IntelliSense, debugging, linting, version control, and more.

To install Visual Studio Code, you will need Administrator Privileges and Internet connectivity.

Installing Visual Studio Code ...

9. Finally, click Finish to complete the installation.



Anaconda3 5.3.0 (64-bit) Setup

Thanks for installing Anaconda3!

Anaconda is the most popular Python data science platform.

Share your notebooks, packages, projects and environments on Anaconda Cloud!

☐ Learn more about Anaconda Cloud

☐ Learn how to get started with Anaconda

At this point, the anaconda is installed in the Windows machine and is ready to be used. To verify the install the run the following commands in the terminal and check their output.

```
python -V
conda list
```

An environment is a isolated set of python binaries and packages created by anaconda and it helps to separate dependencies across different projects.

The following commands allows to list environments, create them, enter environment, install and uninstall packages in a environment, exit environment and delete environment respectively.

```
conda info --envs
conda create --name python365 python=3.6.5
activate python365
conda install requests
conda remove requests
deactivate
conda env remove --name python365
```

1.1.1 Installing Jupyter Notebook using Anaconda

As we've already installed Anaconda in the previous section, we need not install Jupyter separately. Anaconda installs Python, Jupyter Notebook and other packages commonly used in data science applications.

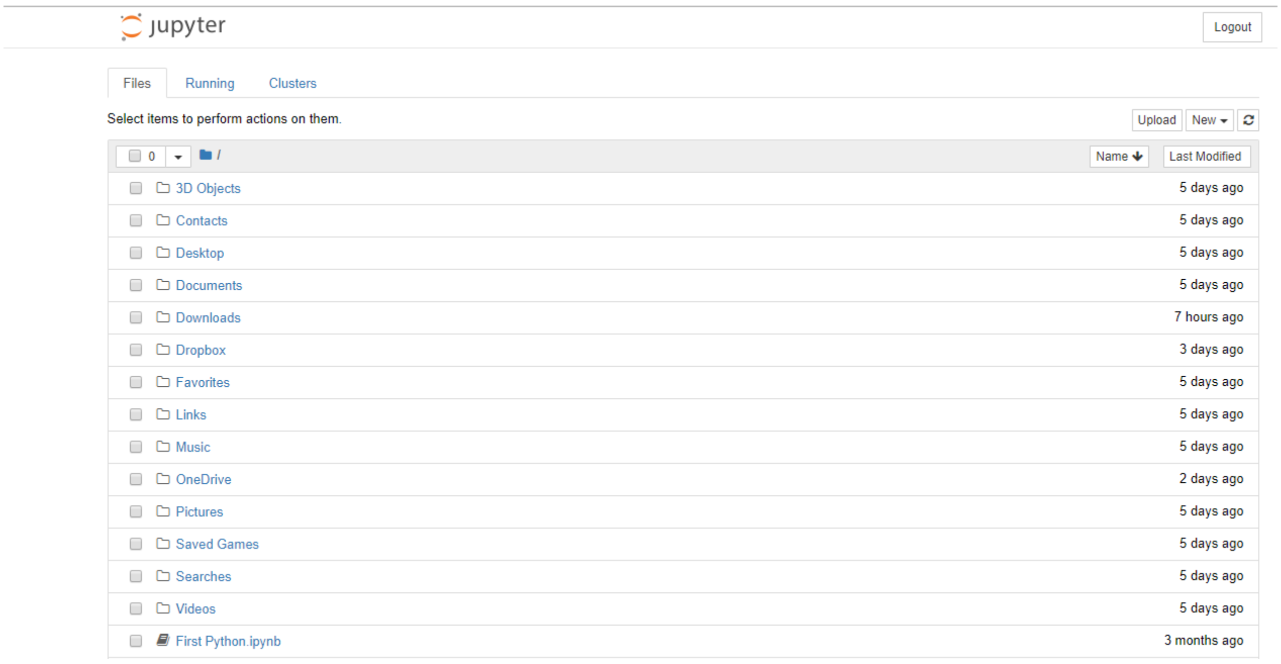
Once you have downloaded and installed Anaconda, the Jupyter Notebook server can be started from the command line using the following command:

```
jupyter notebook
```

On running this from the command line you'll get the following output:

```
[I 17:53:41.751 NotebookApp] Serving notebooks from local directory:
C:\Users\user_name

[I 17:53:41.751 NotebookApp] 0 active kernels

[I 17:53:41.752 NotebookApp] The Jupyter Notebook is running at:

[I 17:53:41.752 NotebookApp]
http://localhost:8888/?token=c233980d78ff505cc3c9d66d012d282b0056ed93c69fa
81b

[I 17:53:41.753 NotebookApp] Use Control-C to stop this server and shut
down all kernels (twice to skip confirmation).
```

The Notebook Dashboard will look like this:



Select items to perform actions on them.

Name	Last Modified
3D Objects	5 days ago
Contacts	5 days ago
Desktop	5 days ago
Documents	5 days ago
Downloads	7 hours ago
Dropbox	3 days ago
Favorites	5 days ago
Links	5 days ago
Music	5 days ago
OneDrive	2 days ago
Pictures	5 days ago
Saved Games	5 days ago
Searches	5 days ago
Videos	5 days ago
First Python.ipynb	3 months ago

New notebook files can then be created from this dashboard.

1.1.2 Installing Spyder using Anaconda

Spyder is a scientific computing environment for Python and is written in Python. Spyder can be integrated with many of the scientific computing packages like NumPy, SciPy, Pandas, IPython, etc. Like other packages, Spyder is shipped along with Anaconda.

To install the package with conda, run the following command:

```
conda install -c anaconda spyder
```

1.2. Functions, First-class Functions and Immutable Data

In this section, we will discuss functions, first-class functions and immutable data.

1.2.1 Functions

A function is a block of code that can be reused several times to do the same set of action or related action based on the parameters passed to it. There are several built-in functions like *print*, *open*, etc. In addition to that, own functions can be created, commonly known as user-defined functions. The statement to create a function is called as function definition.

The rules to define a function is as follows.

1. A function block should always begin with a **def** keyword followed by the function name and then the parameters covered by a opening and closing parenthesis.
2. The parameters are optional if they are assigned with a default value. When the number of parameters expected are unknown, a variable name prefixed by single asterisk and double asterisk symbol can be used. ***posArgs** denotes positional arguments passed as a list and ****kwArgs** denotes keyword arguments passed as a dictionary.
3. An unassigned string can optionally be the first statement of the function called as **docstring** of the function. It is used to describe the purpose of a function.
4. The code block should be indented. The optional **return** keyword is used to exit the function with required values to the function caller. If this keyword doesn't return anything or is not at all used, then the function returns **None** by default.

Syntax:

```
def function_name(paramA, paramB='defaultValue', *posArgs, **kwArgs):
   '''This is the docstring that describes the function purpose'''
   # Any python expressions and statements can be placed here
   return someValueA, someValueB
```

The function can be called as shown in the example below.

```
myPosArgs = ['posArg0value', 'posArg1value', 'posArg2value']
myKwArgs = {'key0': 'value0', 'key1': 'value1'}
function_name('paramAvalue', *myPosArgs, key3='value3', **myKwArgs)
```

1.2.2 First-class Functions

"Functions are First class citizens in Python" is a common phrase in python and it means, functions are also objects just like other data types like string, integer, list, etc. In other words, functions can be assigned to variables, stored in data types like list, dict, etc., and passed as arguments to other functions.

Example:

1.2.2.1 Functions are Objects

```
def greet(name):
  return 'Hello ' + name
print(type(greet))
print(dir(greet))
```

The above example displays the fact that any defined function is an object of the class “function” and has set of attributes associated with it.

1.2.2.2 Functions can be Assigned to Variables

```
def greet(name):
  return 'Hello ' + name

print(greet('World'))
new_greet = greet
print(new_greet('World'))
```

The above code illustrates re-assigning a function to another variable just like any other object.

1.2.2.3 Functions can be Passed as Parameters

```
def louder(word):
   return word.upper()

def whisper(word):
   return word.lower()

def greet(name, func):
   return 'Hello ' + func(name)

print(greet('World', louder))
print(greet('World', whisper))
```

An example for passing functions as arguments is given above.

1.2.2.4 Functions can be Stored in Other Data Structures

```
def greet(name):
        return 'Hello ' + name
    my_list = [greet]
    print(my_list[0]('World'))
```

Above example demonstrates storing a function in other data types like list.

1.2.3 Immutable Data

Since everything in python is an object, their mutability i.e. ability to be changed/modified depends on the type of the object. A list of immutable data types are given in the following table.

Class	**Description**	**Immutable?**
bool	Boolean Value	✓
int	Integer	✓
float	Floating-point number	✓
list	Mutable sequence of objects	✗
tuple	Immutable sequence of objects	✓
str	Character string	✓
set	Unordered set of distinct objects	✗
dict	Associative mapping or dictionary	✗

The built-in function *id* used to identify the memory pointer of any python object and with that, some examples of mutable and immutable data types are given below.

```
x = 1
y = x
x += 1
print(id(x))
print(id(y))
print(x)
print(y)
```

The above example prints the identifier of both *x* and *y*. It can be understood from the output that both pointers are different and thereby showing that integer data type is immutable. The same can be done for any other immutable data types such as *str*, *tuple*, *float* and *tuple*.

```
x = [1, 2, 3]
y = x
x.append(4)
print(id(x))
print(id(y))
print(x)
print(y)
```

The above example shows the mutability of the list data type. Other mutable data types like dict, set works the same way. Mutability matters because it helps in writing efficient programs. When two strings are concatenated, a new third string is created instead of modifying any one of them since strings are immutable by nature.

There are some exceptions in immutability. Consider a tuple having 2 elements, first one be a string and the second element be a list. Though the tuple itself is immutable, the second element of the tuple i.e. the list is mutable. So it can be said that the bindings of the immutable objects are unchangeable but the objects they are bound to is still changeable.

When the mutability and the functions are involved together, new terms arises such as **pass by value** and **pass by reference**. When a immutable object is passed to a function as a parameter, it is passed by value whereas in case of a mutable object, the parameter just holds the reference. This is explained by the following example.

Example:

```
def modifyList(my_list):
   my_list.append(3)
   return my_list

print('List')
old_list = [1, 2]
print(id(old_list))
new_list = modifyList(old_list)
print(id(new_list))
```

```
def modifyNumber(number):
   number += 5
   return number

print('Integer')
old_number = 5
print(id(old_number))
new_number = modifyNumber(old_number)
print(id(new_number))
```

1.3. Iterators and Generators

In this section, we will discuss about iterators and generators.

1.3.1 Iterators

It is a python object that allows to be used with a "for … in …:" loop i.e. traverse through a collection of elements as decided by an iterator. The objects of built-in data types such as list, dict, tuple and set are some examples of the iterables. Any object that need to be an iterable should implement the following two methods in its class.

- __iter__ Returns the iterator object itself
- __next__ Returns next value and a StopIteration error once everything is looped.

The above requirement is known as iterator protocol.

Example:

```
class Counter(object):
   def __init__(self, low, high):
       self.current = low
       self.high = high

   def __iter__(self):
       return self

   def __next__(self):
       if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

print('\na')
a = Counter(5, 10)
for i in a:
    print(i, end=' ')

print('\nb')
b = Counter(3, 4)
print(next(b), end=' ')
print(next(b), end=' ')
print(next(b), end=' ')
```

The above code example illustrates the use of __*iter*__ and __*next*__ methods of the iterable class. The built-in function *iter* returns an iterator by taking an iterable object as its parameter.

Example:

```
mystring = "mystring"
string_iterator = iter(mystring)
print(string_iterator.__next__())
print(string_iterator.__next__())
print(list(string_iterator))
```

1.3.2 Generators

Generators allows to define functions that acts like iterators. It is easy to implement and is possible by the python keyword *yield*. These generator functions can yield as many values as they want, even infinite and will be yielding a value in their each turn.

Generators simplifies the creation of iterators. We can say that all generators are iterators but not all iterators are generators. Because, a generator is a function that returns a sequence of values than a single value.

Example:

```
def counter_generator(low, high):
   while low <= high:
      yield low
      low += 1

for i in counter_generator(5,10):
   print(i, end=' ')
print('\na')
a = counter_generator(3, 4)
print(next(a), end=' ')
print(next(a), end=' ')
print(next(a), end=' ')
```

The above example explains the use of *yield* keyword and using the generator expressions in *for..in..* loop as well as using the *next* built-in function.

Generator expressions are similar to list comprehensions except that the former returns a generator object whereas the latter returns a list object. The syntax for generator expression is that the for loop iteration should be enclosed within a parenthesis.

Example:

```
squares = (i*i for i in range(10))

print(squares.__next__())
print(next(squares))
print(next(squares))

print('\nPrinting from for loop:')
for i in squares:
   print(i)

print('\nCalling next function after the generator end\n')
print(next(squares))
```

The above example demonstrates creating a generator expression and iterating over the generator object. Even conditionals can be added to generator expressions as given below.

Example:

```
pal = (i for i in map(str, range(1000)) if i == i[::-1] and i[-1] == '5')
for i in pal:
   print(i)
```

The above program prints all the palindrome numbers that ends with 5 within 1000.

1.4. Working with Collections

Collections is a standard builtin module in python that has several additional container data types. These are built on top of primary container data types like list, dict, set and tuple. There are many functions available in python for processing these collections and let us cover the immutable ones since our focus will be on functional programming in python.

The collection functions used here should ensure that the state management is localized to the iterator object created as a part of loop evaluation.

1.4.1 Reductions

These are functions used to reduce the given collection to a single final value. Generally, the resulting final value will be an aggregation of the given collection.

Some of the reduction functions are given below.

1.4.1.1 any()

This function takes an iterable as an input and loops through the element. Even if anyone of the element in the iterable is *True*, then this function returns *True*. Else *False* is returned.

It is equivalent to the following.

```
def any(iterable):
   for element in iterable:
       if element:
           return True
   return False
```

Example:

Write a program that checks whether at least one of the given number is an even number.

```
numbers = [7, 91, 25, 43, 64, 37]
iterable = [ i % 2 == 0 for i in numbers]
if any(iterable):
   print('The given numbers has at least one even number')
else:
   print('The given numbers does not have an even number at all')
```

1.4.1.2 all()

Similar to *any()* function, this also takes an iterable as the parameter. It returns *True* only if all the elements in the given iterable is *True*. Else *False* is returned by this function.

It is equivalent to the following.

```
def all(iterable):
   for element in iterable:
       if not element:
           return False
   return True
```

Example:

Write a program that checks whether all the given numbers are even numbers.

```
numbers = [2, 4, 6, 8, 10]
iterable = [ i % 2 == 0 for i in numbers]
if all(iterable):
   print('All the given numbers are even numbers')
else:
   print('All the given numbers are not even numbers')
```

1.4.1.3 sum()

This function takes an iterable as an input parameter and optionally takes a *start* parameter. If *start* parameter is not given explicitly, then 0 is taken as the *start* parameter. The type of the elements in the iterable should always be an integer or float.

The given iterable is looped and each element of the iterable is added to the *start* value until all elements are looped. Finally, the sum of all these elements is returned by this function.

Example:

```
numbers = [1, 2, 3, 4, 5]
sum_of_numbers = sum(numbers)
print(sum_of_numbers)
```

```
sum_of_numbers_with_offset = sum(numbers, 10)
print(sum_of_numbers_with_offset)
```

1.4.1.4 len()

This function returns the total number of elements in an iterable given as the input parameter. A sequence or a collection that implements iterable such as string, tuple, list, dictionary, set can be given as the input to this function.

Example:

```
numbers = [10, 20, 30, 40, 50]
total_numbers = len(numbers)
print(total_numbers)
```

1.4.1.5 max()

This functions takes an iterable as an input and returns the largest element in the given iterable. Optionally, it can take a *key* and a *default* value when a dictionary is passed to the function.

Example:

```
numbers = [11, 2, 33, 404, 55]
maximum_number = max(numbers)
print(maximum_number)
```

1.4.1.6 min()

This functions takes an iterable as an input and returns the smallest element in the given iterable. Optionally, it can take a *key* and a *default* value when a dictionary is passed to the function.

Example:

```
numbers = [11, 2, 33, 404, 55]
minimum_number = min(numbers)
print(minimum_number)
```

1.4.1.7 reduce()

This function requires two parameters as input. The first parameter should be a function itself and the second parameter should be an iterable. Since this is not a built-in function, it need to be imported from the built-in module “functools”.

The function passed in the parameter should take two parameters i.e. the first one holds the accumulated value and the second one has the next element of the iterable. While initializing, the first two elements of the iterable is passed to the function.

It is roughly equivalent to the following.

```
def reduce(function, iterable, initializer=None):
   it = iter(iterable)
   if initializer is None:
       value = next(it)
   else:
       value = initializer
   for element in it:
       value = function(value, element)
   return value
```

Example:

```
from functools import reduce

def concatenate(a, b):
   return str(a) + str(b)

numbers = [1, 2, 3, 4, 5]
concatenated_string = reduce(concatenate, numbers)
print(concatenated_string)
```

1.4.2 Mappings

These are functions used to transform the elements in a given collection. Sorting the elements, reversing the order of the elements, providing a sequence number to each element are some examples of transformation. A custom function can also be passed to do a transformation of your choice. The size of the collection returned by the Mapping functions will be same as the size of the collection passed as input.

1.4.2.1 map()

This function is used to modify each element of a given iterable. It takes a function as the first parameter, an iterable as the second parameter and returns an iterator at the end. On looping through the returned iterator, the given function will be applied on each element and the output of the function will be returned instead of the original element.

Example:

```
def greet(a):
   return 'Hi ' + str(a)

numbers = [1, 2, 3, 4, 5]
mapped_numbers = map(greet, numbers)
print(list(mapped_numbers))
```

1.4.2.2 zip()

This function is used to aggregate elements from each of the given iterable i.e. interleaves values from the given ‘n’ iterators. It takes ‘n’ iterators as input parameters and returns an iterator of tuples where the ‘n’-th tuple contains the ‘n’-th element from each iterable given to the function.

The given iterator of tuples stops when the smallest given iterable is completed i.e. the length of the returned iterator will be equal to the length of the given smallest iterable.

It is equivalent to the following code.

```
def zip(*iterables):
   sentinel = object()
   iterators = [iter(it) for it in iterables]
   while iterators:
       result = []
       for it in iterators:
           elem = next(it, sentinel)
           if elem is sentinel:
               return
           result.append(elem)
       yield tuple(result)
```

Example:

```
number1 = [2, 4, 6, 8]
number2 = [3, 6, 9, 12]
number3 = [5, 10, 15]

zipped_list = list(zip(number1, number2, number3))
print(zipped_list)

new_number1, new_number2, new_number3 = zip(*zipped_list)
print(new_number1, new_number2, new_number3)
```

1.4.2.3 sorted()

This functions sorts the elements in the given iterable and returns the sorted list. Optionally, it takes two more parameters i.e. *key* to transform the given element before sorting and *reverse* to determine the sorting order such as ascending/descending.

Example:

```
numbers = [35, 23, 14, 64, 55]
sorted_numbers = sorted(numbers)
print(sorted_numbers)

strings = ['HiJ', 'kLm', 'Abc', 'mnO', 'eFg', 'Pqr']
sorted_strings = sorted(strings, key=str.lower, reverse=True)
print(sorted_strings)
```

1.4.2.4 reversed()

This functions takes an iterable as an input parameter and returns a new iterator that reverses the order of the elements in the given iterable.

Example:

```
numbers = [1, 2, 3, 4, 5]
reversed_numbers = list(reversed(numbers))
print(reversed_numbers)
```

1.4.2.5 enumerate()

This function takes an iterable as an input parameter and returns an iterator of tuples. The zeroth element of the tuple will have the sum of the offset value and the index of the iterable element and the first element of the tuple will be the iterable element itself.

It is equivalent to the following code.

```
def enumerate(sequence, start=0):
   n = start
   for elem in sequence:
       yield n, elem
       n += 1
```

Example:

```
colours = ['red', 'black', 'white', 'yellow', 'blue', 'green']
enumerated_colours = list(enumerate(colours, 1))
print(enumerated_colours)
```

1.4.3 Filter

These functions are used to reduce the size of the given collection by passing a condition. Based on the condition, the filter functions will reject or allow the elements and finally return a shrunken collection.

1.4.3.1 filter()

This function takes a function and an iterable as the input parameters and returns an iterator. The size of the returned iterator will be less than the given iterable if at least any one of the element fails to be true in the given iterable. Only the elements that are *True* to the applied function will be passed onto the returned iterator.

It is equivalent to the following generator expression.

```
def filter(function, iterable):
   return (item for item in iterable if function(item))
```

Example:

```
def is_even(number):
   return number % 2 == 0
```

```python
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(is_even, numbers))
print(even_numbers)
```

1.5. Higher-order Functions - I

A function that accepts functions as parameters or has functions as the return type is called as Higher-order functions. In fact, we have already used this higher-order functions in the last section itself. *sorted*, *map*, *reduce*, *filter*, etc., are some examples of the higher-order functions that takes some functions as the parameters.

For a function to be higher-order function, it must be a first-class function i.e. the functions should be treated as any other objects such that they can be assigned to variables, stored in lists or dictionaries, etc.,

There is a built-in module named "*functools*" in python which is specifically meant for higher-order functions. This module provides functions that can act on any other functions or any callable objects such as a lambda function or the methods of the class objects.

1.5.1 Loops vs. Recursions

Loops and Recursions are different ways to iterate over a given set of inputs. A block of code provided for iteration will be evaluated in each iteration.

Functional programming is possible with both loops as wells as recursions. Though it may seem obvious that loops use mutable data types and recursions use immutable data types, it is worth noting that a loop can be done without mutable data types (such as list and dict) and similarly, a recursion can be done using mutable data.

1.5.2 Loops

Loop is a set of instructions executed sequentially till a given condition is true. A typical Loop performs a condition check, jump across elements and carry out an increment/decrement operation.

A loop can be considered as an recursion in terms of functionality but the implementation is vastly different. Main difference is that, loop holds a state in a variable to know the current element being iterated. Since it maintains a state of the current iteration and is mutable, it is against functional programming.

1.5.3 Recursion

Recursion is a process of breaking the given set of elements into minimal parts and solving them at some level i.e. an instruction will given to recursion along with a condition just like a loop but instead of a variable holding a state, the stack frames of the programming language will be used.
Recursion involves pushing stack frames to the runtime (for every element), executing given set of instructions, returning a value, and popping back from the stack. As a final result, the given problem will be solved and reduced to one single value (the return value of the first function call). Unlike Loops, Recursions doesn't use mutable data types since everything is being stored in the stack frames. This allows us to write programs that operate on higher levels of abstraction.

Our interest will be in Recursion rather than loops since recursion allows to use multiple processor cores at the same time which in turn ends in a greater cumulative CPU clock speed. In case of loops, it uses same core throughout the iteration and the clock speed of a CPU core cannot be increased more than a limit and becomes too expensive with greater clock speeds. In Big Data, this clock speed plays crucial role in data processing and hence Recursion is our point of interest.

1.5.3.1 Head Recursion

Recursions can be done in two ways, i.e., Head and Tail recursion. A head recursion is one in which the recursive call is made at the beginning of the function and then, the instruction to process the input is carried out.

In this method, each recursive call freezes its local variables and then makes the further recursive calls till the given condition is reached. For greater inputs, the memory will be full with a number of stack frames holding several frozen local variables and is prone to Stack Frame OverFlow error.

Example:

```python
def factorial(number):
   if number == 0:
       return 1
   return number * factorial(number - 1)
```

```
print(factorial(5))
```

Here, the variable *number* is getting stored in every stack frame of the recursive calls and in some point the memory will be full leading to Stack Overflow error. The computation is done at the end here i.e. at the final recursive call the number 1 will be returned and multiplied by popping out each stack frame and it means, for every input, some state is stored in the stack frame till final call and the computation occurs from the top to bottom in the stack. This makes obvious that it is prone to memory error and poor utilization of the memory.

1.5.3.2 Tail Recursion

If a recursive call is the last thing done in a function, then it is called as Tail Recursion. In a way, a tail recursion is similar to a loop. That means, computation will be done along with the every recursive steps and the computed value will be passed on to the next recursive call. Hence, the local variables of the previous stack frames will no longer be necessary and can be dumped from the runtime. This kind of recursion saves from the Stack Overflow error since stack itself gets dumped once the computation is over.

Example:

```
def factorial(number, accumulator=1):
   if (number == 0):
       return accumulator
   return factorial(number - 1, number * accumulator)

print(factorial(5))
```

Calling *factorial* function is the last instruction executed in this function before returning. Also, the computed value in the variable *accumulator* is passed on to every recursive calls, thus makes the stack frame discardable.

1.5.4 “Functools” Module

This builtin module in python is specific for the higher order functions i.e. the functions that can manipulate other functions.

1.5.4.1 lru_cache()

This is a decorator to wrap functions with a memoizing callables i.e. when a function wrapped by this decorator is called for the second time with same set of parameters, then instead of computing the function's instruction all over again, a memoized value is returned. In other words, when wrapped functions with some parameters are called for the first time, the return value of the functions are saved in the memory and for all further identical calls, the saved value will be returned.

LRU stands for “Least Recently Used”. It accepts two parameters, *maxsize* to limit the number of recent return values to cache and *typed* to cache functions with different parameters separately. By default, *maxsize* is set to 128 and *typed* is set to False.

Example:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
   if n < 2:
       return n
   return fibonacci(n-1) + fibonacci(n-2)

print([fibonacci(n) for n in range(16)])
print(fibonacci.cache_info())
```

In the above example, the *fibonacci* function is wrapped with the *lru_cache* decorator. So whenever the function is called for the first time, its return value will be cached. All further calls will be returned from the cache. The last print statement will print the number of hits, misses and caches of the fibonacci function.

1.5.4.2 partial()

This function takes another function as the first parameter followed by positional arguments and keyword arguments as second and third parameter. The provided positional and keyword arguments are applied to the function and returned. This returned function will have arguments applied to it partially.

Example:

```
from functools import partial
```

```
def add(a, b):
   return a + b

add_2 = partial(add, 2)
print(add_2(3))
```

In the above example, partial is called with *add* function and an argument 2. The return value of partial which is an function by itself is assigned to *add_2*. Again, *add_2* is called with parameter 3 which returns 5 as the final answer.

1.5.4.3 partialmethod()

This function is similar to *partial* except that it is used in class methods. The passed func parameter should be a callable or a descriptor.

Example:

```
from functools import partialmethod

class Calculator:
   def add(self, a, b):
       return a + b
   add_2 = partialmethod(add, 2)

calc = Calculator()
print(calc.add_2(3))
```

1.5.4.4 wraps()

This function is used to modify parameters passed to a function before calling it and modify the return value before it is returned. In simple words, this wraps a function with the given function.

Example:

```
from functools import wraps

def add_exclamation(f):
   @wraps(f)
   def wrapper(*args, **kwargs):
        return f(*args, **kwargs) + '!'
    return wrapper

@add_exclamation
def greet(name):
    return 'Hi ' + name

print(greet('User'))
```

1.5.5 “Itertools” Module

This builtin module in python provides several functions to build iterators for different purposes which helps in creating solutions easily and efficiently.

1.5.5.1 count()

Creates an iterator that returns evenly spaced values starting from zero by default. The starting number and the steps can be passed as parameters to this function.

It is roughly equivalent to the following.

```
def count(start=0, step=1):
    number = start
    while True:
        yield number
        number += step
```

1.5.5.2 cycle()

Creates an iterator from the given iterable and after the end of the iterable, the elements from the beginning are yielded all over again. This same process is repeated indefinitely.

It is equivalent to the following.

```
def cycle(iterable):
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

1.5.5.3 repeat()

This function creates an iterator that yields the given object over and over again indefinitely. It optionally accepts a *times* parameter to limit the number of repetitions.

It is equivalent to the following code.

```
def repeat(object, times=None):
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

1.5.5.4 accumulate()

Creates an iterator that returns a list of accumulated sums or accumulated values of any other binary functions (functions accepting two parameters) passed to it.

It is roughly equivalent to the following code.

```
def accumulate(iterable, func=operator.add):
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

1.5.5.5 chain()

This function accepts multiple iterables as the parameter and returns an iterator that yields elements from the first iterable, then proceeds to the next iterable and does the same till all elements in all iterables are yielded. In simple words, it returns an iterator that merges the given iterables.

It is roughly equivalent to the following code.

```
def chain(*iterables):
   for it in iterables:
       for element in it:
           yield element
```

1.5.5.6 chain.from_iterable()

Similar to the *chain()* function, except that instead of taking multiple iterables in the parameters, it takes one single parameter which should be a list of iterables.

It is roughly equivalent to the following code.

```
def from_iterable(iterables):
   for it in iterables:
       for element in it:
           yield element
```

1.5.5.7 compress()

Creates an iterator that returns the elements for whose corresponding selector elements are evaluated to *True*.

It is roughly equivalent to the following code.

```
def compress(data, selectors):
   return (d for d, s in zip(data, selectors) if s)
```

1.5.5.8 dropwhile()

Creates an iterator that drops the elements in the given iterable as long as the given predicate is true and then returns all the further elements as it is.

It is roughly equivalent to the following code.

```
def dropwhile(predicate, iterable):
   iterable = iter(iterable)
   for x in iterable:
       if not predicate(x):
           yield x
           break
   for x in iterable:
       yield x
```

1.5.5.9 filterfalse()

Creates an iterator that returns elements in iterable for which the given predicate is evaluated to *False*.

It is roughly equivalent to the following code.

```
def filterfalse(predicate, iterable):
   if predicate is None:
       predicate = bool
   for x in iterable:
       if not predicate(x):
           yield x
```

1.5.5.10 starmap()

Creates an iterator that applies the given function with each element of the given iterable as parameters. The main difference between *map* and *starmap* is that the arguments are passed a positional arguments directly to the function in map whereas the arguments are grouped into tuples and it is passed as a single argument in starmap.

It is roughly equivalent to the following code.

```
def starmap(function, iterable):
   for args in iterable:
       yield function(*args)
```

1.5.5.11 takewhile()

Creates an iterator that returns the elements from the given iterable as long as the given predicate gets evaluated to *True*.

It is roughly equivalent to the following code.

```
def takewhile(predicate, iterable):
   for x in iterable:
       if predicate(x):
           yield x
       else:
            break
```

1.5.5.12 zip_longest()

Creates an iterator that aggregates elements from each of the given iterable. For iterables with different length, all the missing elements are filled with *None* values.

It is roughly equivalent to the following code.

```
def zip_longest(*args, **kwds):
    fillvalue = kwds.get('fillvalue')
    counter = len(args) - 1
    def sentinel():
        nonlocal counter
        if not counter:
            raise ZipExhausted
        counter -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

1.5.5.13 product()

Creates an iterator that returns the cartesian product of the iterables given in the input.

It is roughly equivalent to the following.

```
def product(*args, repeat=1):
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

1.5.5.14 permutations()

Creates an iterator that returns permutations of the elements in the given iterable. The size of the permutations can be controlled by passing a second parameter to the function.

It is roughly equivalent to the following code.

```
def permutations(iterable, r=None):
   pool = tuple(iterable)
   n = len(pool)
   r = n if r is None else r
   for indices in product(range(n), repeat=r):
       if len(set(indices)) == r:
           yield tuple(pool[i] for i in indices)
```

1.6. Higher-order Functions - II

1.6.1 Callables

In python, functions are not the only things that are callable. In implementation, everything in python is an object and callable is just an attribute of that object identified by the name "*__call__*".

1.6.1.1 "Callable" Function

We can whether an object is a callable or not by using the built-in function "*callable*" as given below.

Example:

```
def greet(name):
   return 'Hi ' + name
print(callable(greet))
myname = 'User'
print(callable(myname))
```

1.6.1.2 "*__call__*" method

All functions are internally objects whose class has a "*__call__*" method in it. To test this out, one can simply call a function like the following.

Example:

```
def greet(name):
  return 'Hello ' + name
print(greet.__call__('World'))
```

1.6.1.3 Classes can be Callables

By default, all Classes will be callable which will return an instance of the class. Calling a class directly will internally invoke a method called "*__init__*" if exists. This method is also called as Constructor of the class. A custom return value of the "*__init__*" method is forbidden in python and the instance of the class is always returned.

An instance of a class is not callable unless the class has a "*__call__*" method in it. The following example explains the "*__init__*" method and the "*__call__*" method.

Example:

```
class Greet:
   def __init__(self, greeting):
       self.greeting = greeting

   def __call__(self, name):
       return self.greeting + ' ' + name

greeter = Greet('Hello')
print(greeter.greeting)
print(greeter('World'))
```

1.6.1.4 Types are Callables

The data types in python such as *int*, *float*, *str*, *list*, *tuple* and *dict* are classes by themselves which are callable. This can be verified using the following example.

Example:

```
print(int)
print(float)
print(str)
print(list)
print(tuple)
print(dict)

print(int(123))
print(int.__call__(123))
print(int.__call__)
```

1.6.1.5 Instance Methods are Callables

An object (instance of a class) in python can have attributes and the attributes that are callable is known as “methods” of that class/object. List is a class which has a method named “append” in it. This method is callable because of the “*__call__*” method present in it.

Example:

```
my_list = [1, 2, 3]
print(type(my_list))
print(my_list)

my_list.append(4)
print(my_list)

my_list.append.__call__(5)
print(my_list)

print(my_list.append.__call__)
```

1.6.1.6 Pure Functions

A function is said to be pure if it does not modify the state of anything beyond its scope. This property is important in functional programming where there is no state at all. Think of a list declared and initialized with some values. Assume a function accepts a list object as a parameter, appends some elements and returns the list object. Here the function is modifying the state of the list object which is declared beyond the scope of the function and hence, this is not a pure function. The built-in function “*id*” returns the pointer of the given object.

Example:

```
def append_4(input_list):
   input_list.append(4)
```

```
    return input_list

my_list = [1, 2, 3]
print(id(my_list))

new_list = append_4(my_list)
print(id(new_list))

print(my_list)
print(new_list)
```

In the above example, the function *append_4* is modifying the state of the given object and hence it is an impure function. An example for pure function is given below.

Example:

```
def append_4(input_list):
    temp_list = input_list[:]
    temp_list.append(4)
    return temp_list

my_list = [1, 2, 3]
print(id(my_list))

new_list = append_4(my_list)
print(id(new_list))

print(my_list)
print(new_list)
```

In this example, the function *append_4*, instead of modifying the given object, it clones it using slicing operator, makes changes to the cloned object and returns the clone. Thus, the function doesn't modify anything declared beyond its scope and can be considered as pure function. Pure functions makes the program easy to understand for the readers. Avoiding global variables in programming is a general thumb rule and likewise, avoiding state changes to the objects outside the scope is a thumb rule for functional programming.

1.6.1.7 Classes are Mutable

Though functional programming can be done with python, it is not purely functional because everything in python is an object and objects are mutable i.e. they maintain a state invariably. This makes python a general purpose language. Not only objects, even classes are mutable. In other words, a class definition is an object by itself which can be instantiated to other objects by making a call. Modifying a class definition or a function definition during runtime is called as monkey patching and is recommended to avoid such methods unless required in python.

In the following example, the *__call__* method of the class *Greet* is overridden in the runtime with a lambda function. The pointer of the actual *__call__* method and the overridden *__call__* method is different.

Example:

```
class Greet:
   def __init__(self, greeting):
       self.greeting = greeting

   def __call__(self, name):
       return self.greeting + ' ' + name

print(id(Greet.__call__))

Greet.__call__ = lambda self, name: self.greeting + ' ' + name + '!'
print(id(Greet.__call__))

greeter = Greet('Hello')
print(greeter('World'))
```

1.6.2 Currying

Currying is a technique of transforming a function that takes multiple parameters in such a way that its returns a sequence of functions that takes one parameter at a time and finally returning the expected result.

Example:

The function given below accepts two arguments and returns the sum of them as the output.

```
def add(a, b):
   return a + b
print(add(2, 3))
```

After currying, the function accepts *parameter a* initially and returns an another function that accepts *parameter b* which in turn returns the sum of them.

```
def add(a):
   def add_a(b):
       return a + b
   return add_a
print(add(2)(3))
```

Currying functions enables easier reuse of more abstract functions. It allows to specialize the functions or partially apply them and makes it easy for complex programs where different kind of possibilities are expected.

1.6.2.1 Operator Module

In functional programming, it is common to do some arithmetic and comparison operations using python's intrinsic operators and this function might be required in some other functions like *map*, etc. All these operations as functions can be done manually by simply defining the parameters between different operators and returning them. However, if there is a module that has all the functions defined by default and in an efficient way, it reduces plenty of work of the programmer and makes the code clean as well.

The "*operator*" module exports a set of functions to do the actions of the python's intrinsic operators such as +*, -, *, /, //, is, and, not, ==, >=, <=, =, etc*.

Example:

```
from operator import add, sub, mul, truediv, floordiv, ge, le, eq

print('3 + 2 \t-->\t', add(3, 2))
print('3 - 2 \t-->\t', sub(3, 2))
print('3 * 2 \t-->\t', mul(3, 2))
print('3 / 2 \t-->\t', truediv(3, 2))
print('3 // 2 \t-->\t', floordiv(3, 2))
print('3 >= 2 \t-->\t', ge(3, 2))
print('3 <= 2 \t-->\t', le(3, 2))
print('3 == 2 \t-->\t', eq(3, 2))
```

The following table maps all the intrinsic operators of python to their appropriate functions in operator module. (Referred from Python's official documentation)

Operation	**Syntax**	**Function**
Addition	a + b	add(a, b)
Concatenation	seq1 + seq2	concat(seq1, seq2)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	truediv(a, b)
Modulo Division	a // b	floordiv(a, b)
Bitwise And	a & b	and_(a, b)
Bitwise Exclusive Or	a ^ b	xor(a, b)
Bitwise Inversion	~ a	invert(a)
Bitwise Or	a | b	or_(a, b)
Exponentiation	a ** b	pow(a, b)
Identity	a is b	is_(a, b)
Identity	a is not b	is_not(a, b)
Indexed Assignment	obj[k] = v	setitem(obj, k, v)
Indexed Deletion	del obj[k]	delitem(obj, k)
Indexing	obj[k]	getitem(obj, k)
Left Shift	a << b	lshift(a, b)

Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Matrix Multiplication	a @ b	matmul(a, b)
Negation (Arithmetic)	- a	neg(a)
Negation (Logical)	not a	not_(a)
Positive	+ a	pos(a)
Right Shift	a >> b	rshift(a, b)
Slice Assignment	seq[i:j] = values	setitem(seq, slice(i, j), values)
Slice Deletion	del seq[i:j]	delitem(seq, slice(i, j))
Slicing	seq[i:j]	getitem(seq, slice(i, j))
String Formatting	s % obj	mod(s, obj)
Subtraction	a - b	sub(a, b)
Truth Test	obj	truth(obj)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

The operations having in-place versions can also be done using functions in operator module.

Example:

```
from operator import iadd
a = 2
print('a += 3 \t-->\t', iadd(a, 3))
```

These functions are listed in the table below.

Syntax	Function
a += b	iadd(a, b)
a &= b	iand(a, b)
a += b	iconcat(a, b)
a //= b	ifloordiv(a, b)
a <<= b	ilshift(a, b)
a %= b	imod(a, b)
a *= b	imul(a, b)
a @= b	imatmul(a, b)
a |= b	ior(a, b)
a **= b	ipow(a, b)
a >>= b	irshift(a, b)
a -= b	isub(a, b)

a /= b	itruediv(a, b)
a ^= b	ixor(a, b)

1.7. File Operations in Python

File operations are an important concept in any web application. There are several functions in Python for file operations like creating, reading, updating and deleting.

1.7.1 open()

The `open()` function is the primary function for working with files. It takes two parameters, filename and mode. There are four modes of opening a file:

- "r" - Read - Default value. Opens a file for reading. Will throw an error if the file does not exist
- "a" - Append - Opens a file for appending. This operation creates the file if it does not exist
- "w" - Write - Opens a file for writing. This operation creates the file if it does not exist, like the append mode
- "x" - Create - Creates the specified file and throws an error if the file exists.

For opening a file in the working directory, we can simply specify the name of the file. For example, to open a file named ‘abc.txt’, the syntax is as follows:

```
file = open("abc.txt")
```

Assume that the file is saved in the working directory and has the following content:

```
This is a test file.
It is used for this lab only.
```

We can use open() function to read the contents of the file, in read mode and print the content.

```
file = open("abc.txt", "r")
print(file.read())
```

This operation will open the file and print the content as follows:

```
This is a test file.
```

```
It is used for this lab only.
```

We can also specify the number of characters to be read from any particular file. For example, to read the first 10 characters of the file:

```python
file = open("abc.txt", "r")
print(file.read(10))
```

This will give the output: “`This is a `“.

We can read the lines using the readline() method.

```python
file = open("abc.txt", "r")
print(file.readline())
```

This will return the output

```
This is a test file.
```

To loop through each line in the file,

```python
file = open("abc.txt", "r")
for x in file:
  print(x)
```

The output will be as follows:

```
This is a test file.
It is used for this lab only.
```

1.7.2 Writing to an Existing File

To write to an existing file, we need to add one of the two parameters to the open() function:

- "a" - Append - will append to the end of the file
- "w" - Write - will overwrite any existing content

Open the file "abc.txt" and append content to the file:

```python
file = open("abc.txt", "a")
file.write("I have added third line to this file now!")
```

Similarly, to overwrite the existing content,

```
file = open("abc.txt", "w")
file.write("The content is overwritten!")
```

1.7.3 Closing a File

The `close()` method is used to close the file. When this method is used, all buffers are closed and close the file. For example, to close the abc.txt file:

```
file.close()
```

Creating a New File:

To create a new file in Python, we use the open()

- "x" - Create - will create a file, returns an error if the file exist
- "a" - Append - will create a file if the specified file does not exist
- "w" - Write - will create a file if the specified file does not exist

For example, to create a file called, newfile.txt, the syntax is:

```
f = open("newfile.txt", "x")
```

A new empty file gets created by this.

1.8. Data Preprocessing

We'll use Jupyter Notebook for these exercises. For the data preprocessing tasks, we'll use the Pandas library.

1.8.1 Installing and Using Pandas

For installing and using Pandas, NumPy has to be installed in the system. Since we're using the Anaconda stack, Pandas comes as a pre-built package. Pandas, once installed, just has to be imported like the other modules. The syntax for checking the Pandas version after importing it is:

```
import pandas
```

```
pandas.__version__
```

You will get the version of Pandas as the output. NumPy is imported under the alias np and Pandas as pd:

```
import pandas as pd
```

We need to understand Pandas Series, DataFrame and Index objects in Pandas to work with data efficiently.

Pandas Series:

Pandas Series is an unidimensional array of indexed data, that can be created from a list or array as follows:

```
In[1]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
Out[1]: 0 0.25
1 0.50
2 0.75
3 1.00
dtype: float64
```

Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes. The values are simply a NumPy array:

```
In[2]: data.values
Out[2]: array([ 0.25, 0.5 , 0.75, 1. ])
```

The index is an array-like object of type `pd.Index`, as follows:

```
In[3]: data.index
Out[3]: RangeIndex(start=0, stop=4, step=1)
```

From a Series, data can be accessed by the associated index as follows:

```
In[4]: data[1]
Out[4]: 0.5

In[5]: data[1:3]
Out[5]: 1 0.50
```

```
2 0.75
dtype: float64
```

1.8.2 Handling Missing Data

Handling missing data is an important preprocessing step. Real world data is not clean and homogeneous like the data that is presented to us in tutorials. Many of the data sets will have missing data and different data sources indicate missing data in different formats. Missing data is generally referred to as null, NaN or NA.

1.8.2.1 Missing Data in Pandas

The way in which Pandas handle missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.

1.8.2.2 None: Missing Python Data

The first sentinel value used by Pandas is None , a Python singleton object used for missing data in Python code. None is a Python object, and it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects):

```
In[1]: import numpy as np
import pandas as pd

In[2]: vals1 = np.array([1, None, 3, 4])
vals1
Out[2]: array([1, None, 3, 4], dtype=object)
```

Any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:

```
In[3]: for dtype in ['object', 'int']:
            print("dtype =", dtype)
            %timeit np.arange(1E6, dtype=dtype).sum()
            print()
dtype = object
```

```
10 loops, best of 3: 78.2 ms per loop

dtype = int
100 loops, best of 3: 3.06 ms per loop
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a None value, you will generally get an error.

```
In[4]: vals1.sum()
TypeError                                   Traceback (most recent call
last)
<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()
```

1.8.2.3 NaN: Missing Numerical Data

NaN (acronym for Not a Number) is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
Out[5]: dtype('float64')
```

NumPy selected a native floating-point type for this array: this means that unlike the object array, this array supports fast operations pushed into compiled code. Irrespective of the operation, the result of arithmetic with NaN will be another NaN:

```
In[6]: 1 + np.nan
Out[6]: nan

In[7]: 0 * np.nan
Out[7]: nan
```

Aggregates over the values are well defined, but not always useful:

```
In[8]: vals2.sum(), vals2.min(), vals2.max()
Out[8]: (nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

```
In[9]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

```
Out[9]: (8.0, 1.0, 4.0)
```

NaN and None in Pandas

NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

```
In[10]: pd.Series([1, np.nan, 2, None])
Out[10]: 0 1.0
1 NaN
2 2.0
3 NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be upcast to a floating-point type to accommodate the NA:

```
In[11]: x = pd.Series(range(2), dtype=int)
        x
Out[11]: 0 0
         1 1
dtype: int64

In[12]: x[0] = None
        x
Out[12]: 0 NaN
         1 1.0
dtype: float64
```

1.8.2.4 Null Values

Pandas consider both None and NaN as for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures.

They are as follows:

`isnull()` - Generate a Boolean mask indicating missing values

`notnull()` - Opposite of isnull()

`dropna()` - Return a filtered version of the data `fillna()` - Return a copy of the data with missing values filled or imputed

Detection of null values

Pandas data structures have two useful methods for detecting null data: isnull() and notnull() . Either one will return a Boolean mask over the data. For example:

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])
In[14]: data.isnull()
Out[14]: 0 False
         1 True
         2 False
         3 True
dtype: bool

In[15]: data[data.notnull()]
Out[15]: 0 1
        2 hello
dtype: object
```

The isnull() and notnull() methods produce similar Boolean results for Data Frames.

Dropping null values

In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a Series, the result is straightforward:

```
In[16]: data.dropna()
Out[16]: 0 1
        2 hello
dtype: object
```

For a DataFrame , there are more options. Consider the following DataFrame :

```
In[17]: df = pd.DataFrame([[1, np.nan, 2],
                           [2, 3, 5],
                           [np.nan, 4, 6]])
        df
Out[17]:        0  1  2
```

```
            0 1.0 NaN 2
            1 2.0 3.0 5
            2 NaN 4.0 6
```

Single values cannot be dropped from a DataFrame; we can only drop full rows or full columns. By default, dropna() will drop all rows in which any null value is present:

```
In[18]: df.dropna()
Out[18]:        0   1  2
            1 2.0 3.0  5
```

Alternatively, NA values can be dropped along a different axis; `axis=1` drops all columns containing a null value:

```
In[19]: df.dropna(axis='columns')
Out[19]:      2
            0 2
            1 5
            2 6
```

But this method drops some good data also; we can drop rows or columns with all NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is how='any' , such that any row or column (depending on the axis keyword) containing a null value will be dropped. You can also specify `how='all'` , which will only drop rows/columns that are all null values:

```
In[20]: df[3] = np.nan
df
Out[20]:        0   1  2   3
            0 1.0 NaN  2 NaN
            1 2.0 3.0  5 NaN
            2 NaN 4.0  6 NaN
```

```
In[21]: df.dropna(axis='columns', how='all')
Out[21]:        0   1  2
            0 1.0 NaN  2
            1 2.0 3.0  5
            2 NaN 4.0  6
```

Using the thresh parameter we can specify a minimum number of non-null values for the row/column to be kept:

```
In[22]: df.dropna(axis='rows', thresh=3)
Out[22]:        0   1  2   3
            1 2.0 3.0  5 NaN
```

Here the first and last row have been dropped, because they contain only two non-null values.

Filling in Null Values

Instead of dropping NA values, we can replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. Pandas provides the fillna() method, which returns a copy of the array with the null values replaced.

Consider the following Series :

```
In[23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
Out[23]:    a 1.0
            b NaN
            c 2.0
            d NaN
            e 3.0
dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
In[24]: data.fillna(0)
Out[24]:    a 1.0
            b 0.0
            c 2.0
            d 0.0
            e 3.0
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
In[25]: # forward-fill
data.fillna(method='ffill')
Out[25]:     a 1.0
             b 1.0
             c 2.0
             d 2.0
             e 3.0
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
In[26]: # back-fill
data.fillna(method='bfill')
Out[26]:     a 1.0
             b 2.0
             c 2.0
             d 3.0
             e 3.0
dtype: float64
```

1.9. Exploratory Data Analysis

Exploratory Data Analysis (EDA) is about analysing the data at different levels of granularity. EDA is essential for determining the right algorithm for machine learning and further data analysis. In many cases, EDA can reveal patterns that are typically revealed by a data mining algorithm. EDA also helps us to understand data characteristics, so we can choose the right algorithm for the given problem.

This lab requires that the libraries SciPy, NumPy, Pandas and Matplotlib are already installed.

In this lab, the following dataset has been used understanding the basic operations while doing EDA.

http://archive.ics.uci.edu/ml/machine-learning-databases/optdigits/optdigits.tra

In this exercise, we'll first import the data and do a basic description, we'll then query or index data to have a deeper understanding. We'll also do some do some pattern identification by using pattern identification.

Importing the Data

- Import the Pandas library as pd.
- The data in csv format is then read by passing the URL in which the data is found.
- Print the data to see what is there.

```
import pandas as pd
digits = pd.read_csv("http://archive.ics.uci.edu/ml/machine-learning-
databases/optdigits/optdigits.tra",
                     header=None)
print(digits)
```

Your output will be like this:

```
0   1   2   3   4   5   6   7   8   9  ...  55  56  57  58  59  60  61  \
0      0   1   6  15  12   1   0   0   0   7 ...   0   0   0   6  14   7   1
1      0   0  10  16   6   0   0   0   0   7 ...   0   0   0  10  16  15   3
2      0   0   8  15  16  13   0   0   0   1 ...   0   0   0   9  14   0   0
3      0   0   0   3  11  16   0   0   0   0 ...   0   0   0   0   1  15   2
4      0   0   5  14   4   0   0   0   0   0 ...   0   0   0   4  12  14   7
5      0   0  11  16  10   1   0   0   0   4 ...   3   0   0  10  16  16  16
6      0   0   1  11  13  11   7   0   0   0 ...   0   0   0   1  13   5   0
7      0   0   8  10   8   7   2   0   0   1 ...   0   0   0   4  13   8   0
8      0   0  15   2  14  13   2   0   0   0 ...   0   0   0  10  12   5   0
9      0   0   3  13  13   2   0   0   0   6 ...   0   0   0   3  15  11   6
10     0   0   6  14  14  16  16   8   0   0 ...   0   0   0  10  12   0   0
11     0   0   0   3  16  11   1   0   0   0 ...   0   0   0   0   2  14  14
12     0   0   0   4  13  16  16   3   0   0 ...   0   0   0   0   5  15   4
13     0   0   7  12   6   2   0   0   0   0 ...   0   0   0   5  16   9   0
14     0   0   7  11  11   6   0   0   0   9 ...   0   0   0  14  16  12  10
15     0   1  10  15   8   0   0   0   0   6 ...   0   0   0   9  15   8   0
16     0   0   0   1  11   7   0   0   0   0 ...   0   0   0   0   3  15   0
17     0   0   5  12  16  16   3   0   0   0 ...   0   0   0   8  12   0   0
18     0   0   1   8  13  13   2   0   0   4 ...   0   0   0   1  13  12   4
19     0   0   0   2  13  12   4   0   0   0 ...   0   0   0   0   0  15   3
20     0   0   4  11  15  16  15   0   0   0 ...   0   0   0   6  14   2   0
21     0   0   4  10  13  11   1   0   0   2 ...   0   0   0   6  13  11   1
22     0   0   3  11  13  14   6   0   0   0 ...   0   0   0   3  13  10   0
23     0   0   1   4  11  13   7   0   0   2 ...   0   0   0   0   1  14   3
```

```
24      0   0   9  13   1   0   0   0   0   0 ...   5   0   0   4  15  16  16
25      0   0   9  16  11   0   0   0   0   4 ...   1   0   0  10  16   9   9
26      0   0   2  13   9   0   0   0   0   0 ...   0   0   0   3  16  14   4
27      0   0   0  10  12   0   0   0   0   0 ...   0   0   0   1  11  14  12
28      0   0   0   0  10  13   0   0   0   0 ...   0   0   0   0   0   8  15
29      0   0   7   9  13  11   2   0   0   6 ...   0   0   0  13  12   8   1
...    ..  ..  ..  ..  ..  ..  ..  ..  ..  .. ...  ..  ..  ..  ..  ..  ..  ..
3793    0   0  15  15  16  14   1   0   0   3 ...   0   0   0  15  16  12   0
3794    0   0   2  13  16  15   4   0   0   0 ...   0   0   0   3  14   1   0
3795    0   0  12  16   7   0   0   0   0   2 ...   0   0   0  11  16  16  16
3796    0   0   0   3  13   0   0   0   0   0 ...   0   0   0   0   6  12   0
3797    0   0   0  12   8   0   0   0   0   0 ...   0   0   0   0  14   6   0
3798    0   0   0   9  12   0   0   0   0   0 ...   0   0   0   0  10  11   0
3799    0   0   5  16  13   1   0   0   0   0 ...   0   0   1   6  10  12  12
3800    0   0   6  16   8   0   0   0   0   2 ...   0   0   0   7  13  16  13
3801    0   0   5  16  11   0   0   0   0   1 ...   0   0   0   4  14  12   2
3802    0   1  12  16  14   4   0   0   0   8 ...   0   0   0  13  15  15  10
3803    0   0   3  14  13   3   0   0   0   0 ...   0   0   0   3  13  13   6
3804    0   0  12  16  16  10   1   0   0   0 ...   0   0   1  10  16  16  16
3805    0   0   0   8  11   0   0   0   0   0 ...   0   0   0   0   7  12   0
3806    0   1   9  16  14   6   0   0   0   5 ...   0   0   0  11  16   7   0
3807    0   0   4  14  15   3   0   0   0   0 ...   0   0   0   3  14  14   6
3808    0   2  15  16  15   1   0   0   0   3 ...   0   0   0  14  13  16  11
3809    0   0   5  15  16   6   0   0   0   0 ...   0   0   0   5  15  16  15
3810    0   0   4  16  11   1   0   0   0   0 ...   0   0   0   2  12  16  11
3811    0   0   0   0   7  11   1   0   0   0 ...   0   0   0   2   0   7  11
3812    0   0   0   6  13   0   0   0   0   0 ...   0   0   0   0   6   9   0
3813    0   0   0   6   8   0   0   0   0   0 ...   0   0   1   4   9   8   0
3814    0   0   9  16   6   0   0   0   0   2 ...   0   0   0  10  13   1   0
3815    0   0   9  16  12   1   0   0   0   3 ...   0   0   0   8  16  16  16
3816    0   1  10  16  16   4   0   0   0   8 ...   0   0   2  13  16  12   5
3817    0   0   6  16  11   0   0   0   0   1 ...   0   0   1   7  14  16  12
3818    0   0   5  13  11   2   0   0   0   2 ...   0   0   0   8  13  15  10
3819    0   0   0   1  12   1   0   0   0   0 ...   0   0   0   0   4   9   0
3820    0   0   3  15   0   0   0   0   0   0 ...   0   0   0   4  14  16   9
3821    0   0   6  16   2   0   0   0   0   0 ...   0   0   0   5  16  16  16
3822    0   0   2  15  16  13   1   0   0   0 ...   0   0   0   4  14   1   0
```

```
      62                                 63                         64
0                                       0                   0                   0
1                                       0                   0                   0
2                                       0                   0                   7
3                                       0                   0                   4
4                                       0                   0                   6
5                                   16                    6                     2
6                                       0                   0                   5
7                                       0                   0                   5
8                                       0                   0                   0
9                                       0                   0                   8
10                                   0                    0                     7
11                                   1                    0                     1
12                                   0                    0                     9
13                                   0                    0                     5
14                                   1                    0                     3
15                                   0                    0                     0
16                                   0                    0                     4
17                                   0                    0                     7
18                                   0                    0                     8
19                                   0                    0                     4
20                                   0                    0                     7
21                                   0                    0                     8
22                                   0                    0                     5
23                                   0                    0                     9
24                                  16               16                         1
25                                13                    6                       2
26                                   0                    0                     0
27                                   1                    0                     6
28                                   2                    0                     1
29                                   0                    0                     8
...                                 ..                   ..                    ..
3793                           0                        0                       5
3794                           0                        0                       7
3795                     16                          8                          2
3796                           0                        0                       4
3797                           0                        0                       4
3798                           0                        0                       4
```

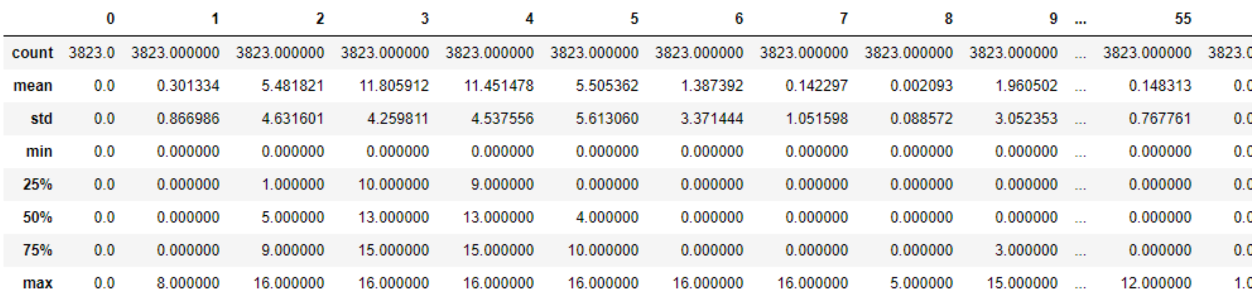
```
3799                    2                    0                    9
3800                    9                    0                    9
3801                    0                    0                    0
3802                    2                    0                    3
3803                    0                    0                    0
3804                    7                    0                    3
3805                    0                    0                    4
3806                    0                    0                    8
3807                    0                    0                    0
3808                    1                    0                    3
3809                    1                    0                    0
3810                    1                    0                    1
3811                    0                    0                    1
3812                    0                    0                    4
3813                    0                    0                    4
3814                    0                    0                    8
3815                    8                    0                    9
3816                    0                    0                    3
3817                    1                    0                    9
3818                    1                    0                    9
3819                    0                    0                    4
3820                    0                    0                    6
3821                    5                    0                    6
3822                    0                    0                    7

[3823 rows x 65 columns]
```

1.9.1. Describing the data

The process of EDA starts by getting a basic description of the data using the `describe()` function.

```
digits.describe()
```

Your output will be something like this:



	0	1	2	3	4	5	6	7	8	9	...	55	
count	3823.0	3823.000000	3823.000000	3823.000000	3823.000000	3823.000000	3823.000000	3823.000000	3823.000000	3823.000000	...	3823.000000	3823.
mean	0.0	0.301334	5.481821	11.805912	11.451478	5.505362	1.387392	0.142297	0.002093	1.960502	...	0.148313	0.
std	0.0	0.866986	4.631601	4.259811	4.537556	5.613060	3.371444	1.051598	0.088572	3.052353	...	0.767761	0.
min	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.
25%	0.0	0.000000	1.000000	10.000000	9.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.
50%	0.0	0.000000	5.000000	13.000000	13.000000	4.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.
75%	0.0	0.000000	9.000000	15.000000	15.000000	10.000000	0.000000	0.000000	0.000000	3.000000	...	0.000000	0.
max	0.0	8.000000	16.000000	16.000000	16.000000	16.000000	16.000000	16.000000	5.000000	15.000000	...	12.000000	1.

8 rows × 65 columns

As you see, the function returns the count, mean, standard deviation, minimum and maximum values and the quantiles of the data.

We can inspect the first and last set of rows using the head() and tail() functions. Example, we can inspect the first and last 10 rows of the digits dataset using the following code:

```
#inspecting the first 10 rows
first = digits.head(10)
print(first)
```

Output

```
   0   1   2   3   4   5   6   7   8   9  ...  55  56  57  58  59  60  61  62
\
0  0   1   6  15  12   1   0   0   0   7 ...   0   0   0   6  14   7   1   0
1  0   0  10  16   6   0   0   0   0   7 ...   0   0   0  10  16  15   3   0
2  0   0   8  15  16  13   0   0   0   1 ...   0   0   0   9  14   0   0   0
3  0   0   0   3  11  16   0   0   0   0 ...   0   0   0   0   1  15   2   0
4  0   0   5  14   4   0   0   0   0   0 ...   0   0   0   4  12  14   7   0
5  0   0  11  16  10   1   0   0   0   4 ...   3   0   0  10  16  16  16  16
6  0   0   1  11  13  11   7   0   0   0 ...   0   0   0   1  13   5   0   0
7  0   0   8  10   8   7   2   0   0   1 ...   0   0   0   4  13   8   0   0
8  0   0  15   2  14  13   2   0   0   0 ...   0   0   0  10  12   5   0   0
9  0   0   3  13  13   2   0   0   0   6 ...   0   0   0   3  15  11   6   0

   63                                                                     64
0                                      0                                   0
1                                      0                                   0
2                                      0                                   7
3                                      0                                   4
4                                      0                                   6
5                                      6                                   2
6                                      0                                   5
```

```
7                                    0                                    5
8                                    0                                    0
9   0   8
[10 rows x 65 columns]
```

```
#inspecting the last 10 rows
last = digits.tail(10)
print(last)
```

Output

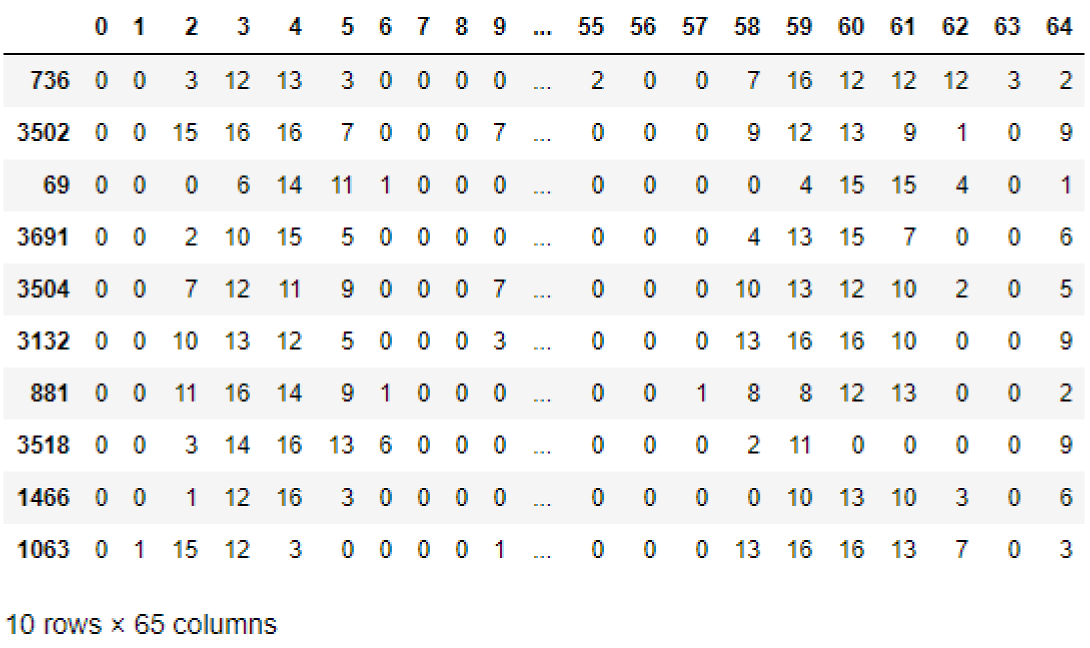
```
      0   1   2   3   4   5   6   7   8   9  ...  55  56  57  58  59  60  61  \
3813  0   0   0   6   8   0   0   0   0   0 ...   0   0   1   4   9   8   0
3814  0   0   9  16   6   0   0   0   0   2 ...   0   0   0  10  13   1   0
3815  0   0   9  16  12   1   0   0   0   3 ...   0   0   0   8  16  16  16
3816  0   1  10  16  16   4   0   0   0   8 ...   0   0   2  13  16  12   5
3817  0   0   6  16  11   0   0   0   0   1 ...   0   0   1   7  14  16  12
3818  0   0   5  13  11   2   0   0   0   2 ...   0   0   0   8  13  15  10
3819  0   0   0   1  12   1   0   0   0   0 ...   0   0   0   0   4   9   0
3820  0   0   3  15   0   0   0   0   0   0 ...   0   0   0   4  14  16   9
3821  0   0   6  16   2   0   0   0   0   0 ...   0   0   0   5  16  16  16
3822  0   0   2  15  16  13   1   0   0   0 ...   0   0   0   4  14   1   0

      62                               63                                64
3813                       0                        0                     4
3814                       0                        0                     8
3815                       8                        0                     9
3816                       0                        0                     3
3817                       1                        0                     9
3818                       1                        0                     9
3819                       0                        0                     4
3820                       0                        0                     6
3821                       5                        0                     6
3822                       0                        0                     7
[10 rows x 65 columns]
```

1.9.2. Sampling the data

If the dataset is fairly large, we can take a sample out of it using the `sample()` function as follows:

```
digits.sample(10)
```

This will generate a sample of 10 from the given dataset as follows:



	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	64
736	0	0	3	12	13	3	0	0	0	0	...	2	0	0	7	16	12	12	12	3	2
3502	0	0	15	16	16	7	0	0	0	7	...	0	0	0	9	12	13	9	1	0	9
69	0	0	0	6	14	11	1	0	0	0	...	0	0	0	0	4	15	15	4	0	1
3691	0	0	2	10	15	5	0	0	0	0	...	0	0	0	4	13	15	7	0	0	6
3504	0	0	7	12	11	9	0	0	0	7	...	0	0	0	10	13	12	10	2	0	5
3132	0	0	10	13	12	5	0	0	0	3	...	0	0	0	13	16	16	10	0	0	9
881	0	0	11	16	14	9	1	0	0	0	...	0	0	1	8	8	12	13	0	0	2
3518	0	0	3	14	16	13	6	0	0	0	...	0	0	0	2	11	0	0	0	0	9
1466	0	0	1	12	16	3	0	0	0	0	...	0	0	0	0	10	13	10	3	0	6
1063	0	1	15	12	3	0	0	0	0	1	...	0	0	0	13	16	16	13	7	0	3

10 rows × 65 columns

We can create a random index and get random rows from the DataFrame. In the following code, the random package is used, that has a module called sample, using which we can sample the data, in combination with `range()` and `len()` functions. We can also use `iloc` to select the exact rows that we want to include in the sample.

```
import numpy as np
from random import sample
randomIndex = np.array(sample(range(len(digits)), 10))
digitsSample = digits.iloc[randomIndex]
print(digitsSample)
```

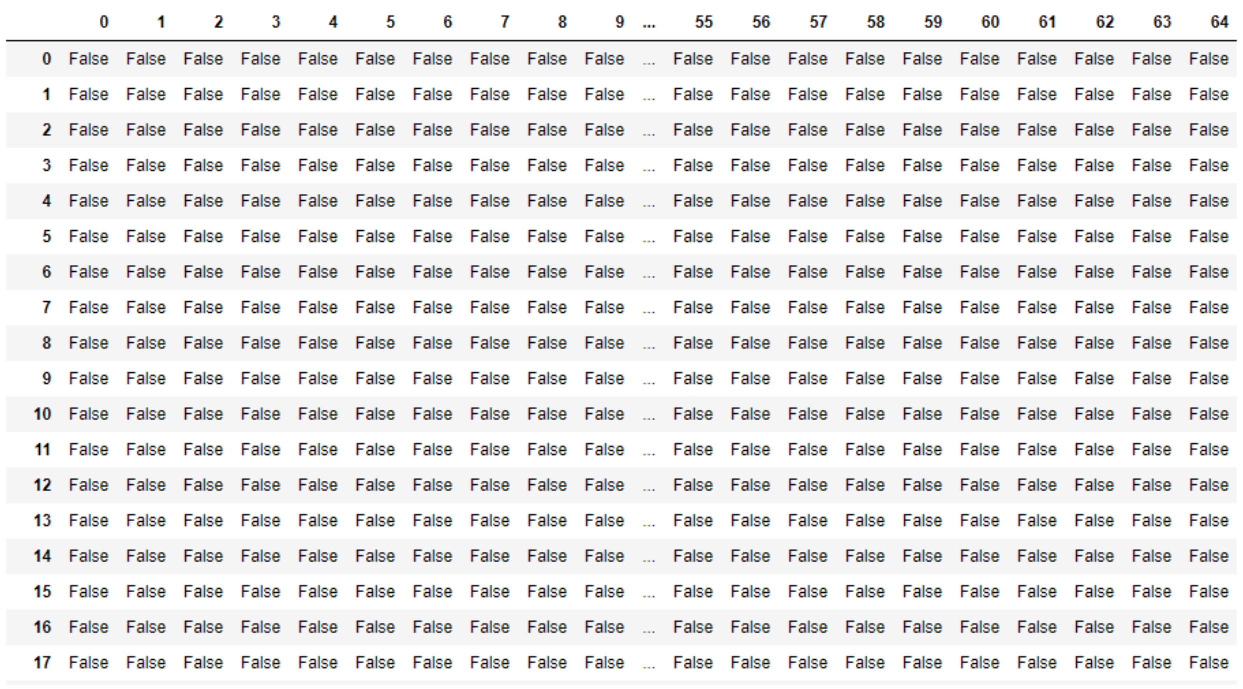
The output will be as follows:

```
      0   1   2   3   4   5   6   7   8   9 ...  55  56  57  58  59  60  61
\
1069   0   2  10  15   8   0   0   0   0   6 ...   0   0   0  12  16  14  11
1523   0   0   0   9   4   0   0   0   0   0 ...   0   0   0   0   7  12   0
1697   0   0   3  10  13   4   0   0   0   0 ...   0   0   0   4  16  16  10
1402   0   2  10  16  16  16   7   0   0  14 ...   0   0   0  15   7   0   0
```

```
2249   0   0  11  16  10   1   0   0   0   0 ...   0   0   0   8  15  14  11
3754   0   0   6  12   7   0   0   0   0   2 ...   0   0   0   3   7  12  12
308    0   0   3  15   3   0   0   0   0   0 ...   0   0   0   5  16   4   0
1442   0   0   0   5  15   0   0   0   0   0 ...   0   0   0   0   6  11   0
3727   0   0   5   9  13   8   0   0   0   0 ...   0   0   0  10  12   3   0
3265   0   3  16  15   3   0   0   0   0   8 ...   1   0   2  14  16  15  12

     62                                 63                                64
1069                    0                       0                          9
1523                    0                       0                          4
1697                    0                       0                          9
1402                    0                       0                          5
2249                    3                       0                          9
3754                    3                       0                          8
308                        0                      0                        4
1442                    0                       0                          4
3727                    0                       0                          4
3265                 12                       1                            2

[10 rows x 65 columns]
```

1.9.3. Identifying Missing Values

To identify the missing values in the dataset, the `isnull()` function can be used. In the result, True or False appears in each cell.
True indicates that the value contained within the cell is a missing value, False indicates that the cell contains a ‘normal’ value. In our case, the output will be something like this:



	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	64
0	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
5	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
7	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
8	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
9	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
10	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
11	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
12	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
13	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
14	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
15	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
16	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
17	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False

This indicates that the dataset is complete without any missing values.

In real-world this will not be the case. We'll come across a lot of missing values. In those cases, we can either delete the whole record or keep the ones where some desired features are still available.

We can also use some imputation methods to fill in missing values. We've seen about filling in missing data in the previous section, like `fillna()`, `dropna()`, etc.

1.9.4. Data Analysis

For data analysis and visualization, we'll use a different dataset called the Iris dataset, the commonly used dataset in machine learning projects. The dataset is available here: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/.

Data description of the iris dataset will give the following output:

```
       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
```

1.9.5. Queries

Querying the data will give an even more closer outlook towards the data. The query() function is used to test some hypotheses. The following code checks two hypotheses - “Is the petal length usually greater than the sepal length?” or “Is the petal length sometimes equal to the sepal length?”.

```
# Hypothesis 1
iris.query('Petal_length > Sepal_length')

# Hypothesis 2
iris.query('Petal_length == Sepal_length')
```

The code will produce the following output:

```
Empty DataFrame
Columns: [Sepal_length, Sepal_width, Petal_length, Petal_width, Class]
Index: []
```

The tested hypotheses do not hold true, and as we see, it will produce an empty dataframe. The following query is also valid and will produce the same output.

```
iris[iris.Petal_length > iris.Sepal_length]
```

1.9.6. Features of the Dataset

Feature engineering is used to increase the predictive power of the algorithms.
We can encode categorical variables into numerical ones using the factorize() feature. Consider the following code:

```
# Factorize the values
labels,levels = pd.factorize(iris.Class)

# Save the encoded variables in “iris.Class”
iris.Class = labels

# Print out the first rows
iris.Class.head()
```

The code will produce the following output:

```
0    0
```

```
1    0
2    0
3    0
4    0
Name: Class, dtype: int64
```

The Random Forest algorithm can be used to find the important features. This algorithms randomly generates thousands of decision trees and finds out the best fit model. This is used to find out how better or worse a model performs when one variable is left out. You can use the Scikit-Learn Python library to implement this algorithm:

```
# Import ‘RandomForestClassifier’
from sklearn.ensemble import RandomForestClassifier

# Isolate Data, class labels and column values
X = iris.iloc[:,0:4]
Y = iris.iloc[:,-1]
names = iris.columns.values

# Build the model
rfc = RandomForestClassifier()

# Fit the model
rfc.fit(X, Y)

# Print the results
print("Features sorted by their score:")
print(sorted(zip(map(lambda x: round(x, 4), rfc.feature_importances_),
names), reverse=True))
```

The code will produce the following output:

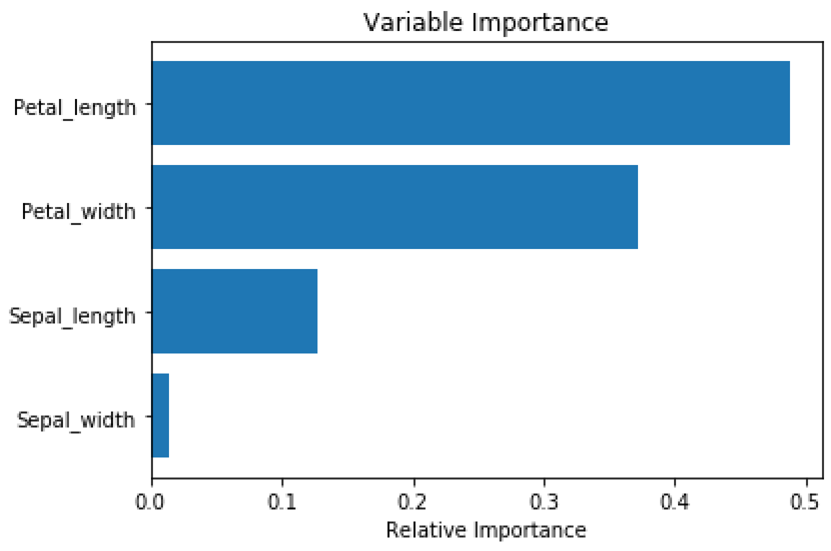
```
Features sorted by their score:
[(0.51910000000000001, 'Petal_length'), (0.35120000000000001,
'Petal_width'), (0.11269999999999999, 'Sepal_length'),
(0.016899999999999998, 'Sepal_width')]
```

The result shows that the feature set with petal length and petal width is the best feature set.

The result can also be visualized using matplotlib as follows:

```
# Import ‘pyplot’ and ‘numpy’
import matplotlib.pyplot as plt
import numpy as np

# Isolate feature importances
importance = rfc.feature_importances_

# Sort the feature importances
sorted_importances = np.argsort(importance)

# Insert padding
padding = np.arange(len(names)-1) + 0.5

# Plot the data
plt.barh(padding, importance[sorted_importances], align='center')

# Customize the plot
plt.yticks(padding, names[sorted_importances])
plt.xlabel("Relative Importance")
plt.title("Variable Importance")

# Show the plot
plt.show()
```

The output will be as follows:



1.10. Curve Fitting

For curve fitting we need the library SciPy. The scipy package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

1.10.1 The SciPy Library

SciPy comes along with Anaconda. It can be conveniently installed using the command:

```
conda install -c anaconda scipy
```

Some of the specific submodules in SciPy are as follows:

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package

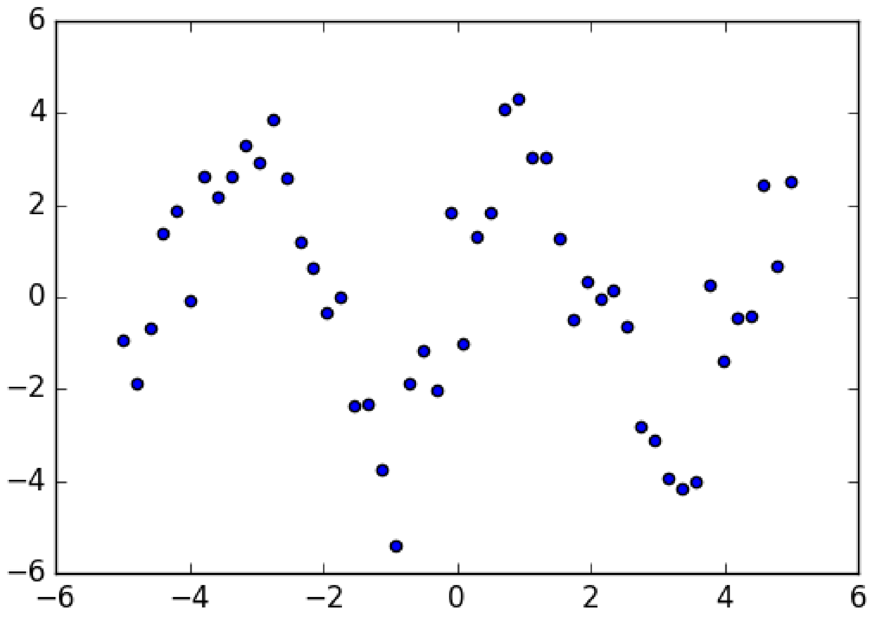
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions
scipy.stats	Statistics

All these submodules depend on NumPy, but are independent of each other. The syntax for importing NumPy and SciPy modules is:

```
import numpy as np
from scipy import stats #same for all the submodules
```

1.10.2 Example for Simple Curve Fitting

```
import numpy as np

#First seed the random number generator for reproducibility
np.random.seed(0)

x_data = np.linspace(-5, 5, num=50)
y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=50)

# And plot it
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data)
```



Now fit a simple sine function to the data

```
from scipy import optimize

def test_func(x, a, b):
    return a * np.sin(b * x)

params, params_covariance = optimize.curve_fit(test_func, x_data, y_data,
                                               p0=[2, 2])

print(params)
```

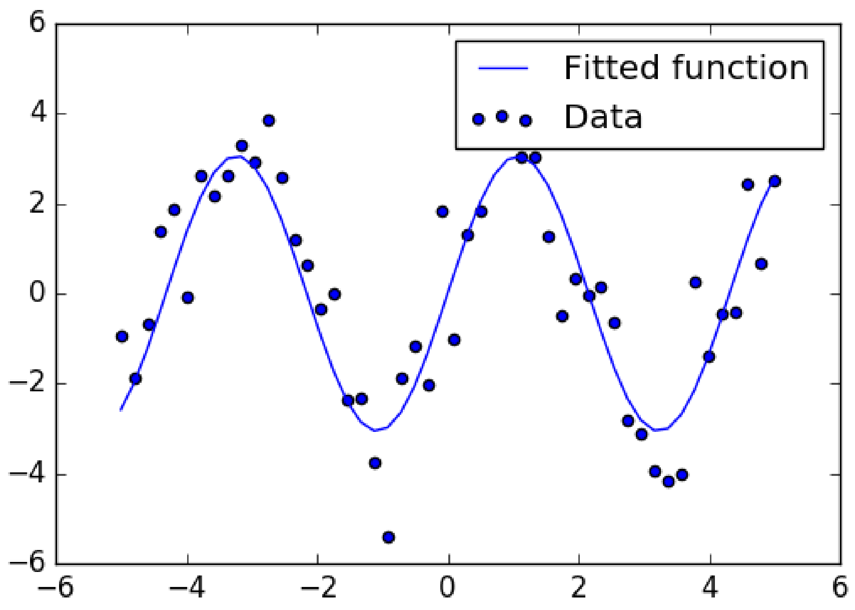
Output:

```
[3.05931973 1.45754553]
```

And plot the resulting curve on the data

```
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_data, test_func(x_data, params[0], params[1]),
         label='Fitted function')
```

```
plt.legend(loc='best')

plt.show()
```


Fitted function
Data