

ASSIGNMENT

Course Code	20CSE421A
Course Name	Data Science Foundation
Programme	B. Tech.
Department	Computer Science & Engineering
Faculty	Faculty of Engineering & Technology

Name of the Student	Subhendu Maji
Reg. No	18ETCS002121
Semester/Year	7 th semester / 2018 batch
Course Leader/s	

Declaration Sheet			
Student Name	Subhendu Maji		
Reg. No	18ETCS002121		
Programme	B. Tech.	Semester/Year	7 th sem /2018 batch
Course Code	20CSE421A		
Course Title	Data Science Foundation		
Course Date		to	
Course Leader			
<p>Declaration</p> <p>The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of university regulations and will be dealt with accordingly.</p>			
Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			
Signature of the Course Leader and date		Signature of the Reviewer and date	

Declaration Sheet	ii
Contents	iii
Marking Scheme.....	4
Question No. 1	6
A1.1 Python language and its usage in data science	6
A2 Python syntax and core constructs	7
A1.3 Functions: Namespaces and scopes.....	8
A4.1 Exception handling.....	9
A1.5 Libraries for Data Science Applications.....	13
A6 Data modelling, processing.....	15
Question No. 2	16
B1.1 Design a system for reading data in text format using pandas library	16
B1.2 Design a system for storing data in text format using pandas library	19
B1.3 Design about the task in a parallel system using Celery to obtain square root of a value	21
B1.4 Design about the client and server role in a parallel system using Celery to obtain square root of a value	22
Question No. 3	25
B2.1 Implement Python code for reading data in text format using pandas library	25
B2.2 Implement Python code for storing data in text format using pandas library	27
B2.3 Implement Python code for the task in a parallel system using Celery to obtain square root of a value	28
B2.4 Implement Python code for the client and server role in a parallel system using Celery to obtain square root of a value.....	30

Assignment - 01					
Register No		18ETCS002121	Name of Student		Subhendu Maji
Sections		Marking Scheme	Marks		
			Max Marks	First Examiner Marks	Moderat or Marks
Part-A					
	A 1.1	Python language and its usage in data science	04		
	A 1.2	Python syntax and core constructs	06		
	A 1.3	Functions: Namespaces and scopes	05		
	A 1.4	Exception handling	04		
	A1.5	Libraries for Data Science Applications	05		
	A1.6	Data modelling, processing	06		
		Part-A Max Marks	30		
Part B 1					
	B 1.1	Design a system for reading data in text format using pandas library	7		
	B 1.2	Design a system for storing data in text format using pandas library	8		
	B 1.3	Design about the task in a parallel system using Celery to obtain square root of a value	5		
	B 1.4	Design about the client and server role in a parallel system using Celery to obtain square root of a value	10		
		Part-B1 Max Marks	30		
Part B 2					
	B2.1	Implement Python code for reading data in text format using pandas library	10		
	B2.2	Implement Python code for storing data in text format using pandas library	10		
	B2.3	Implement Python code for the task in a parallel system using Celery to obtain square root of a value	5		

	B2.4	Implement Python code for the client and server role in a parallel system using Celery to obtain square root of a value	15		
		Part-B2 Max Marks	40		
	Total Assignment Marks		100		

Course Marks Tabulation				
Component- CET B Assignment	First Examiner	Remarks	Second Examiner	Remarks
A				
B.1				
B.2				
Marks (Max 50)				
Marks (out of 25)				
<div>Signature of First Examiner</div> <div>Signature of Second Examiner</div>				

Solution to Question No. 1:

A1.1 Python language and its usage in data science

Python Programming Language

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.

Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Applications of python:

- Developing web sites and apps
- Data Analysis
- AI and Machine learning.
- Image processing
- Game development

In many scenarios, Python is the programming language of choice for the daily tasks that data scientists tackle, and is one of the top data science tools used across industries. For data scientists who need to incorporate statistical code into production databases or integrate data with web-based applications, Python is often the ideal choice. It is also ideal for implementing algorithms, which is something that data scientists need to do often.

There are also Python packages that are specifically tailored for certain functions, including pandas, NumPy, and SciPy. Data scientists working on various machine learning tasks find that Python's scikit-learn is a useful and valuable tool. Matplotlib, another one of Python's packages, is also a perfect solution for data science projects that require graphics and other visuals.

Reasons to use python in data science:

- The main reason of using python in data science is that python provides a lot of libraries which can be used for data science applications.
- Python is an open source and it is easy to implement because of its simple syntax.
- Python is used for data science because python contains some libraries like pandas, NumPy which can be used for analyzing the data or modification of data.
- Another important reason for using python in data science is that its syntax is simple so that it can be understood by all classes of people.

A2 Python syntax and core constructs

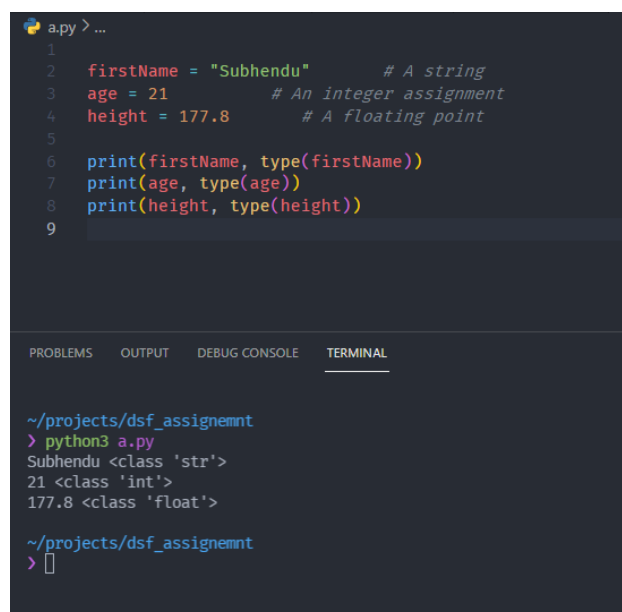
A2.1

Python is completely object oriented, and not "statically typed". We do not need to declare variables before using them, or declare their type. Every variable in Python is an object.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Example:



```
a.py > ...
1
2 firstName = "Subhendu"      # A string
3 age = 21                    # An integer assignment
4 height = 177.8              # A floating point
5
6 print(firstName, type(firstName))
7 print(age, type(age))
8 print(height, type(height))
9

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

~/projects/dsf_assignemnt
> python3 a.py
Subhendu <class 'str'>
21 <class 'int'>
177.8 <class 'float'>

~/projects/dsf_assignemnt
> 
```

Figure 1 python is a dynamically typed language

We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it. A variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location; all the operations done on the variable effects that memory location.

A2.2

As we discussed python is a dynamically typed programming language. So, if we declare the variable without assigning it to value the interpreter will get confused and it will show error.

A2.3

- If we start the name of variable with an underscore, the interpreter will treat it as an inbuilt function.
- In python inbuilt functions start with underscore.
- If the variable name starts with underscore, the interpreter will think that the variable is an inbuilt function.

Rules for creating variables in Python:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive (name, Name and NAME are three different variables).
- The reserved words(keywords) cannot be used naming the variable.

A1.3 Functions: Namespaces and scopes

Namespaces

A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a

Python dictionary. the Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace.

- A namespace in python is a collection of underlying key words and objects that python has within the memory.
- It's a very common concept in object-oriented programming.
- Name space is a key value pair implemented as a dictionary.
- Types of name spaces:
 - Built-in
 - Global
 - Local

Example:

```
var1 = 5
def some_func():
    # var2 is in the local namespace
    var2 = 6
    def some_inner_func():
        # var3 is in the nested local
        # namespace
        var3 = 7
```

Variable Scope

Scope refers to the coding region from which a particular Python object is accessible.

Hence one cannot access any particular object from anywhere from the code, the accessing has to be allowed by the scope of the object.

e.g.

```
def some_func():
    print("Inside some_func")
    def some_inner_func():
        var = 10
        print("Inside inner function, value of var:",var)
    some_inner_func()
    print("Try printing var from outer function: ",var)
some_func()
```

output:

```
Inside some_func
Inside inner function, value of var: 10
```

Traceback (most recent call last):

```
File "/home/1eb47bb3eac2fa36d6bfe5d349dfcb84.py", line 8, in
    some_func()
File "/home/1eb47bb3eac2fa36d6bfe5d349dfcb84.py", line 7, in
some_func
    print("Try printing var from outer function: ",var)
NameError: name 'var' is not defined
```

A4.1 Exception handling

To raise an exception, you use the `raise` statement:

Syntax:

```
raise ExceptionType()
```

The `ExceptionType()` must be subclass of the `BaseException` class. Typically, it is a subclass of the `Exception` class. Note that the `ExceptionType` doesn't need to be directly inherited from the `Exception` class. It can indirectly inherit from a class that is a subclass of the `Exception` class.

The `BaseException` class has the `__init__` method that accepts an `*args` argument. It means that you can pass any number of arguments to the exception object when raising an exception.

The following example uses the `raise` statement to raise a `ValueError` exception. It passes three arguments to the `ValueError` `__init__` method:

```
try:
    raise ValueError('The value error exception', 'x', 'y')
except ValueError as ex:
    print(ex.args)
```

Reraise the current exception

Sometimes, we want to log an exception and raise the same exception again. In this case, we can use the `raise` statement without specifying the exception object.

For example, the following defines a `division()` function that returns the division of two numbers:

```
def division(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError as ex:  
        print('Logging exception:', str(ex))  
        raise
```

If we pass zero to the second argument of the `division()` function, the `ZeroDivisionError` exception will occur. However, instead of handling the exception, we can log the exception and raise it again.

Note that we don't need to specify the exception object in the `raise` statement. In this case, Python knows that the `raise` statement will raise the current exception that has been caught by the `except` clause.

The following code causes a `ZeroDivisionError` exception:

```
division(1, 0)
```

Raise another exception during handling an exception

When handling an exception, we may want to raise another exception. In the `division()` function, we raise a `ValueError` exception if the `ZeroDivisionError` occurs.

If we run the following code, we'll get the detail of the stack trace:

```
1  #!/usr/bin/python3
2
3  def division(a, b):
4      try:
5          return a / b
6      except ZeroDivisionError as ex:
7          raise ValueError('b must not zero')
8
9
10 division(1, 0)
11
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
~/projects/dsf_assignemnt
> python3 a.py
Traceback (most recent call last):
  File "a.py", line 5, in division
    return a / b
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "a.py", line 10, in <module>
    division(1, 0)
  File "a.py", line 7, in division
    raise ValueError('b must not zero')
ValueError: b must not zero

~/projects/dsf_assignemnt
> 
```

- **Try**
 - This block contains one or more statements which are likely to encounter an exception.
 - If the try block gets executed with out any exceptions the except block will be skipped.
- **Except**
 - Except block contains the statements which needs to be executed when we get an exception in the try block.
 - After the execution of except block, the error message will be displayed.
 - We can specify the type of exception in the except block. The except block contains multiple exception blocks. Depending on the type of exception occurred in the try block, the exception block of except will be executed.

- **Finally**
 - First the try block will be executed. If there is any exception occurred in try block then the occurred exception will be checked in the except block. Even if the type of exception is present or absent in the except block, finally will be executed.

A4.2

- Try- Except:
 - At first, try block will be executed. If there is no exception occurred in the try block, the statements in the try block will be execute and the output will be displayed.
 - If some exception occurs in the try block, the respective exception will be executed from the except block.
- Try- Finally:
 - At first, try block will be executed. Even if an exception occurred or there is no exception, finally block will execute for sure.

So, the main difference between try- except and try- finally is that in try except only if there is an exception in try block then except block will be executed. In the case of try- finally, even if there is no exception occurred in the try block, finally block will be executed for sure.

A1.5 Libraries for Data Science Applications

Some essential libraries in Python for data analysis are:

- 1) NumPy
- 2) Pandas
- 3) Tensor flow
- 4) SciPy

NumPy

- It is a Python library for numerical computing on data that can be in the form of large arrays and multi-dimensional matrices.
- These multidimensional matrices are the main objects in NumPy where their dimensions are called axes and the number of axes is called a rank.

- NumPy also provides various tools to work with these arrays and high-level mathematical functions to manipulate this data with linear algebra, Fourier transforms etc.
- Some of the basic array operations that can be performed using NumPy include adding, slicing, multiplying, flattening, reshaping, and indexing the arrays.

Pandas

- Pandas is a Python library for data analysis and data handling.
- Pandas provides various high-performance and easy-to-use data structures and operations for manipulating data in the form of numerical tables.
- Pandas also has multiple tools for reading and writing data between in-memory data structures and different file formats.
- Pandas is perfect for quick and easy data manipulation, data aggregation, reading, and writing the data as well as data visualization. Pandas can also take in data from different types of files such as CSV, excel etc. and create a python object known as data frame.
- A data frame contains rows and columns and it can be used for data manipulation with operations such as join, merge, groupby etc.

Tensor flow

- TensorFlow is an end-to-end open-source platform that has a wide variety of tools, libraries, and resources for Artificial Intelligence.
- You can easily build and train Machine Learning models with high-level API's such as Keras using TensorFlow.
- TensorFlow also allows you to deploy Machine Learning models anywhere such as the cloud, browser, or your own device.

SciPy

- SciPy is a python library for scientific computing and technical computing on the data.
- SciPy library is built on the NumPy array object and it is part of the NumPy stack which also includes other scientific computing libraries and tools such as Matplotlib, pandas etc.

- SciPy allows for various scientific computing tasks that handle data optimization, data integration, data interpolation, and data modification using linear algebra, Fourier transforms, random number generation, special functions, etc.
- Just like NumPy, the multidimensional matrices are the main objects in SciPy, which are provided by the NumPy module itself.

A6 Data modelling, processing

A6.1

- NumPy is extremely fast for binary data loading and storage including support for memory mapped array.
- NumPy is a low-level library but it is extremely fast for data loading and storage.
- Reading text files and other more efficient on disk formats, loading data from databases, and interacting with network sources like web API's.

A6.2

- Mat plot lib and D3 both the libraries are together combined to provide interactive features like zooming and panning.
- You can make a plot in matplotlib, add interactive functionality with plugins that utilize both Python and JavaScript, and then render it with D3.
- mpld3 includes built-in plugins for zooming, panning, and adding tooltips.

A6.3

- Pandas is used for flexible and high-performance group by facility enabling slice and dice and summarize data sets in a natural way.
- Python has some methods which are used to do aggregation on data.

Solution to Question No. 2:**B1.1 Design a system for reading data in text format using pandas library**

Pandas is one of those packages and makes importing and analyzing data much easier.

1. `pandas.read_csv`

Read a comma-separated values (csv) file into DataFrame. Also supports optionally iterating or breaking of the file into chunks. Additional help can be found in the online docs for IO Tools.

Parameter	use
sep	Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator
delimiter	Alias for sep.
header	Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to header=None.
names	List of column names to use. If the file contains a header row, then you should explicitly pass header=0 to override the column names.
index_col	Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.
usecols	Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s).

e.g.

```
import pandas as pd
pd.read_csv("pokemon.csv", header =[1, 2])
```

2. pandas.read_table

Read general delimited file into DataFrame. Also supports optionally iterating or breaking of the file into chunks.

Parameter	use
sep	Delimiter to use. If sep is None, the C engine cannot automatically detect the separator,
header	Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to header=None. Explicitly pass header=0 to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3].
names	List of column names to use. If the file contains a header row, then you should explicitly pass header=0 to override the column names. Duplicates in this list are not allowed.
index_col	Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.
usecols	Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s).

e.g.

```
import pandas as pd
pd.read_table('nba.csv',delimiter=',')
```

3. `pandas.read_fwf`

Read a table of fixed-width formatted lines into DataFrame. Also supports optionally iterating or breaking of the file into chunks. Additional help can be found in the online docs for IO Tools.

Parameter	use
<code>colspecs</code>	A list of tuples giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[).
<code>widths</code>	A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.
<code>infer_nrows</code>	The number of rows to consider when letting the parser determine the colspecs.

4. `pandas.read_clipboard`

Read text from clipboard and pass to `read_csv`.

Parameter	use
<code>sep</code>	A string or regex delimiter. The default of 's+' denotes one or more whitespace characters.

5. `pandas.DataFrame.to_csv`

Write object to a comma-separated values (csv) file.

Parameter	use
<code>sep</code>	String of length 1. Field delimiter for the output file.
<code>na_rep</code>	Missing data representation.
<code>columns</code>	Columns to write.
<code>header</code>	Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.
<code>index</code>	Write row names (index).
<code>encoding</code>	A string representing the encoding to use in the output file, defaults to 'utf-8'.

compression	If str, represents compression mode. If dict, value at 'method' is the compression mode. Compression mode may be any of the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}.
-------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

B1.2 Design a system for storing data in text format using pandas library

1. `sqlite3.connect(database [,timeout ,other optional arguments])`

This API opens a connection to the SQLite database file. We can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds). If the given database name does not exist then this call will create the database. You can specify filename with the required path as well if you want to create a database anywhere else except in the current directory.

2. `cursor.execute(sql [, optional parameters])`

This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style).

For example – `cursor.execute("insert into people values (?, ?)", (who, age))`

3. `connection.execute(sql [, optional parameters])`

This routine is a shortcut of the above execute method provided by the cursor object and it creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.

4. `cursor.executemany(sql, seq_of_parameters)`

This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.

5. `cursor.executescript(sql_script)`

This routine executes multiple SQL statements at once provided in the form of script. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter. All the SQL statements should be separated by a semi colon (;).

6. `connection.commit()`

This method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections.

7. `cursor.fetchall()`

This routine fetch all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available.

To Check Entries in the Database

In sqlite3 we do :

```
cursor = con.execute('select * from student222')
rows = cursor.fetchall()
```

in pandas the same operation can be done :

```
import pandas as pd
pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
```

This is quite a bit of munging that we would rather not repeat each time you query the database.

```
import pandas.io.sql as sql
sql.read_sql('select * from student222', con)
```

pandas also has a `read_sql` function in its `pandas.io.sql` module that simplifies the process.

B1.3 Design about the task in a parallel system using Celery to obtain square root of a value

Parallel programming: defined as an approach in which program data creates workers to run specific tasks simultaneously in a multicore environment without the need for concurrency amongst them to access a CPU. the workers that are sent to perform a task often need to establish communication so that there can be cooperation in tackling a problem. In most cases, this communication is established in such a way that data can be exchanged amongst workers. There are two forms of communication that are more widely known when it comes to parallel programming: shared state and message passing.

Celery is a framework that offers mechanisms to lessen difficulties while creating distributed systems. The Celery framework works with the concept of distribution of work units (tasks) by exchanging messages among the machines that are interconnected as a network, or local workers. A task is the key concept in Celery; any sort of job we must distribute has to be encapsulated in a task beforehand.

It distributes tasks in a transparent way among workers that are spread over the Internet, or local workers. It supports synchronous, asynchronous, periodic, and scheduled tasks.

The first thing we need is a Celery instance. We call this the Celery application or just app for short. As this instance is used as the entry-point for everything you want to do in Celery, like creating tasks and managing workers, it must be possible for other modules to import it.

```
from celery import Celery

app = Celery('tasks', broker='redis://localhost:6379/0')

@app.task
def square_root(x):
    return x ** 0.5
```

The first argument to Celery is the name of the current module. This is only needed so that names can be automatically generated when the tasks are defined in the `__main__` module.

Celery requires a solution to send and receive messages; usually this comes in the form of a separate service called a *message broker*.

The second argument is the broker keyword argument, specifying the URL of the message broker you want to use. Here we are using Redis.

There are several choices available, including: RabbitMQ, Redis and Amazon SQS.

Then, we defined a single task, called `square_root`, which returning the square root of a number.

B1.4 Design about the client and server role in a parallel system using Celery to obtain square root of a value

- The client program, as presented in the above section, have the function of creating and dispatching tasks to the brokers.
- It demonstrates the definition of a task by using the **@app.task** decorator, which is accessible through an instance of Celery application that, for now, will be called `app`.
- There are several types of tasks: synchronous, asynchronous, periodic, and scheduled. When we perform a task call, it returns an instance of type **AsyncResult**.
- The `AsyncResult` object is an object that allows the task status to be checked, its ending, and obviously, its return when it exists. However, to make use of this mechanism, another component, the result backend, has to be active.
- The **Message transport (broker)** is definitely a key component in Celery. Through it, we get to send and receive messages and communicate with workers
- The most complete in terms of functionality are **RabbitMQ** and **Redis**. We will use Redis as a broker as well as result backend.
- A **broker** has the function of providing a means of communication between client applications that send tasks and workers that will execute them. This is done by using task queues. We can have several network machines with brokers waiting to receive messages to be consumed by workers.

- **Workers** are responsible for executing the tasks they have received. Celery displays a series of mechanisms so that we can find the best way to control how workers will behave. We can define the mechanisms as follows: Concurrency mode, Remote control, Revoking tasks
- The result backend component has the role of storing the status and result of the task to return to the client application. From the result backend supported by Celery, we can highlight RabbitMQ, Redis, MongoDB, Memcached, among others.

Setting up the environment

we will set up two virtual environments in Linux. The first one, hostname client, will perform the client role, where app Celery will dispatch the tasks to be executed. The other machine, hostname server, will perform the role of a broker, result backend, and the queues consumed by workers.

Setting up the Client Machine

We will set up a virtual environment with Python 3.8, using the tool pyenv. The goal of pyenv is to not pollute Python present in the operating system with additional modules, but to separate the developing environments necessary for each project.

Now, we have a virtual environment and starting off from the point from where you already installed pip, we will install the necessary packages for our client for e.g., celery.

Setting up the Server Machine

To set up the server machine, we will need redis. We can start a redis-server quickly in a docker container. We will start the redis-server which will listen on port 6379.

Dispatching a simple task

Now we have a ready environment. We can test it by sending a task that will calculate the square root of a value and return a result. First, we must define our task module tasks.py inside the server. we need to import necessary library for our function that will calculate the square root:

e.g.

```
from math import sqrt
from celery import Celery
```

Now, we can create the following instance of Celery, which will represent our client application:

```
app = Celery('tasks', broker='redis://localhost:6379/0')
```

Dispatching a task

Next, we have to set up our result backend, which will also be in Redis, as follows:

```
app.config.CELERY_RESULT_BACKEND = 'redis://localhost:6379/0'
```

then, define our task with the `@app.task` decorator:

```
@app.task
def square_root(value):
    return sqrt(value)
```

At this point, since we have our `tasks.py` module defined, we need to initiate our workers inside the server, where Redis and Celery (with support to Redis) are installed.

```
$ celery -A tasks worker --loglevel=INFO
```

Publish a task

we have a Celery server waiting to receive tasks and send them to workers. The next step is to create an application on the client side to call tasks.

1. We import the Celery class inside the `celery` module, as follows:

```
from celery import Celery
```

2. The next step is to create an instance of the Celery class informing the module containing the `tasks` and then the broker, as done in the server side. This is done with the following code:

```
app = Celery('tasks', broker='redis://192.168.25.21:6379/0')
app.conf.CELERY_RESULT_BACKEND = 'redis://192.168.25.21:6397/0'
```

3. we can create a function to encapsulate the sending of the **`sqr_task(value)`** task. We will create the `manage_sqrt_task(value)` function as follows:

```
def manage_sqrt_task(value):
    result = app.send_task('tasks.sqr_task', args=(value,))
    print(result.get())
```

4. In the `__main__` block, we executed the call to the `manage_sqrt_task(value)` function by passing the input value as 4:

```
if __name__ == '__main__':
    manage_sqrt_task(4)
```


Solution to Question No. 3:

```
In [1]: # import pandas package
import pandas as pd
```

B2.1 Implement Python code for reading data in text format using pandas library**B2.1**

```
In [2]: # contents of comma_input.txt
!cat comma_input.txt
```

```
subhendu, 21, M
Emma, 29, F
James, 35, M
```

```
In [3]: # Reading comma seperated input file to dataframe
# it uses first row of the text file as the default header
pd.read_table("comma_input.txt",delimiter=",")
```

Out[3]:

	subhendu	21	M
0	Emma	29	F
1	James	35	M

```
In [4]: # Reading comma seperated input file to dataframe with default header
pd.read_table("comma_input.txt",delimiter=",", header=None)
```

Out[4]:

	0	1	2
0	subhendu	21	M
1	Emma	29	F
2	James	35	M

```
In [5]: # Reading comma seperated input file to dataframe with custom header names
pd.read_table("comma_input.txt", delimiter=",", names=["name","age","gender"])
```

Out[5]:

	name	age	gender
0	subhendu	21	M
1	Emma	29	F
2	James	35	M

```
In [6]: # contents of space_input.txt
!cat space_input.txt
```

```
Subhendu maji Finance 45000
Henry smith IT 21000
Erica jones Marketing 12750
Andrew Madrid IT 30500
```

```
In [7]: # Reading space seperated input file to dataframe
# it uses first row of the text file as the default header
pd.read_table("space_input.txt", delimiter=" ")
```

Out[7]:

	Subhendu	maji	Finance	45000
0	Henry	smith	IT	21000
1	Erica	jones	Marketing	12750
2	Andrew	Madrid	IT	30500

```
In [8]: # Reading space seperated input file to dataframe with default header
pd.read_table("space_input.txt", delimiter=" ", header=None)
```

Out[8]:

	0	1	2	3
0	Subhendu	maji	Finance	45000
1	Henry	smith	IT	21000
2	Erica	jones	Marketing	12750
3	Andrew	Madrid	IT	30500

```
In [9]: # Reading space seperated input file to dataframe with custom header names
pd.read_table("space_input.txt", delimiter=" ", names=["first_name", "last_name", "department", "salary"])
```

Out[9]:

	first_name	last_name	department	salary
0	Subhendu	maji	Finance	45000
1	Henry	smith	IT	21000
2	Erica	jones	Marketing	12750
3	Andrew	Madrid	IT	30500

B2.2 Implement Python code for storing data in text format using pandas library

B2.2

```
In [10]: # importing sqlite3
import sqlite3
```

```
In [11]: # create table student2121 in database
query = """
        CREATE TABLE student2121
        (course VARCHAR(20), courseID VARCHAR(20),
        MarkObtained REAL, MarkExpected INTEGER);
        """

# create a connection to the SQLite database
con = sqlite3.connect(':memory:')

# execute SQL query
con.execute(query)

# Commit the current transaction
con.commit()
```

```
In [12]: # create entries in table
data = [('Data Science Foundation', 'CSE421A', 48, 50),
        ('Computer Vision', 'CSE407A', 45, 50),
        ('Computational Intelligence', 'CSE422A', 38, 50)]

# query to insert into student2121 table
stmt = "INSERT INTO student2121 VALUES(?, ?, ?, ?)"

# to execute a SQL command repeatedly
con.executemany(stmt, data)

# Commit the current transaction
con.commit()
```

```
In [13]: # get all the rows from table
cursor = con.execute('select * from student2121')

# fetch all rows
rows = cursor.fetchall()

# store the fetched data as a pandas dataframe
pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
```

Out[13]:

	course	courseID	MarkObtained	MarkExpected
0	Data Science Foundation	CSE421A	48.0	50
1	Computer Vision	CSE407A	45.0	50
2	Computational Intelligence	CSE422A	38.0	50

using pandas

```
In [14]: # create a connection to the SQLite database
conn = sqlite3.connect(':memory:')

# column names and records of the table
columns = ["course", "courseID", "MarkObtained", "MarkExpected"]
data = [('Data Science Foundation', 'CSE407A ', 49, 50),
        ('Computer Vision', '20CSE422A ', 45, 50),
        ('Computational Intelligence', 'CSE422A ', 38, 50)]

# create a dataframe
df = pd.DataFrame(data, columns=columns)

# use pandas function to create/write to database
# it creates a new database if it doesnt already exist
# if it exists, we can choose to replace, append or fail
df.to_sql("newstudent2121", conn, if_exists="replace")
```

```
In [15]: # read from sqlite using pandas function
pd.read_sql('select * from newstudent2121', conn)
```

Out[15]:

	index	course	courseID	MarkObtained	MarkExpected
0	0	Data Science Foundation	CSE407A	49	50
1	1	Computer Vision	20CSE422A	45	50
2	2	Computational Intelligence	CSE422A	38	50

B2.3 Implement Python code for the task in a parallel system using Celery to obtain square root of a value

Source code:

```
# tasks.py

from celery import Celery
import math
from time import sleep

app = Celery('tasks', broker='redis://localhost:6379/0',
            backend='redis://localhost:6379/0')

@app.task
def square_root(value):
    sleep(5)
    return f"square root of {value} is {math.sqrt(value)}"
```

To run the application

distracted_liskov

redis

RUNNING

LOGS

INSPECT

STATS

```
1:C 27 Nov 2021 18:56:51.172 # oO0o0O0o0O0o Redis is starting oO0o0O0o0O0o
1:C 27 Nov 2021 18:56:51.172 # Redis version=6.2.6, bits=64, commit=00000000, modified=0, pid=1, just started
1:C 27 Nov 2021 18:56:51.172 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
1:M 27 Nov 2021 18:56:51.172 * monotonic clock: POSIX clock_gettime
1:M 27 Nov 2021 18:56:51.173 * Running mode=standalone, port=6379.
1:M 27 Nov 2021 18:56:51.173 # Server initialized
1:M 27 Nov 2021 18:56:51.173 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
1:M 27 Nov 2021 18:56:51.174 * Ready to accept connections
```

Figure 2 starting redis-server in a docker container

```
~/projects/dsf_assignment
env > celery -A tasks worker --loglevel=INFO

----- celery@LAPTOP-C8GE10C8 v5.2.1 (dawn-chorus)
-----
****
-----
***** Linux-5.10.60.1-microsoft-standard-WSL2-x86_64-with-glibc2.29 2021-11-28 01:53:35
****
-----
** [config]
**
--> app: tasks:0x7fd050e4edf0
**
--> transport: redis://localhost:6379/0
**
--> results: redis://localhost:6379/0
**
-----
****
--> concurrency: 12 (prefork)
-----
*****
--> task events: OFF (enable -E to monitor tasks in this worker)
-----
*****
-----
[queues]
--> celery exchange=celery(direct) key=celery

[tasks]
. tasks.square_root

[2021-11-28 01:53:35,818: INFO/MainProcess] Connected to redis://localhost:6379/0
[2021-11-28 01:53:35,829: INFO/MainProcess] mingie: searching for neighbors
[2021-11-28 01:53:36,853: INFO/MainProcess] mingie: all alone
[2021-11-28 01:53:36,891: INFO/MainProcess] celery@LAPTOP-C8GE10C8 ready.
[2021-11-28 01:53:57,101: INFO/MainProcess] Task tasks.square_root[b3bb6f01-1923-4c91-b818-579f68892c18] received
[2021-11-28 01:54:02,113: INFO/ForkPoolWorker-7] Task tasks.square_root[b3bb6f01-1923-4c91-b818-579f68892c18] succeeded in 5.010767200001283s: 'square root of 16 is 4.0'

```

Figure 3 celery worker running and connected to redis-server

```
~/projects/dsf_assignemnt
> python3
Python 3.8.10 (default, Sep 28 2021, 16:10:42)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from tasks import square_root
>>> result = square_root.delay(16)
>>> result.status
'PENDING'
>>> result.status
'PENDING'
>>> result.status
'SUCCESS'
>>> []
```

Figure 4 output of tasks.py

B2.4 Implement Python code for the client and server role in a parallel system using Celery to obtain square root of a value

Source code of server:

```
# tasks.py
from celery import Celery
import math

app = Celery('tasks', broker='redis://localhost:6379/0',
             backend='redis://localhost:6379/0')

@app.task
def square_root(value):
    return f"square root of {value} is {math.sqrt(value)}"
```

Source code of client (publisher.py):

```
# publisher.py

import sys
from celery import Celery

app = Celery('tasks', broker='redis://localhost:6379/0',
             backend='redis://localhost:6379/0')

promise = app.send_task('tasks.square_root', args=[int(sys.argv[1])])

print(promise.get())
```

To run the application

```
~/projects/dsf_assignemnt
env > celery -A tasks worker --loglevel=INFO

----- celery@LAPTOP-C8GE10C8 v5.2.1 (dawn-chorus)
-- ***** -----
-- ***** ----- Linux-5.10.60.1-microsoft-standard-WSL2-x86_64-with-glibc2.29 2021-11-28 00:42:37
-- *** --- * ---
-- ** ----- [config]
-- ** ----- .> app: tasks:0x7f4a5b031df0
-- ** ----- .> transport: redis://localhost:6379/0
-- ** ----- .> results: redis://localhost:6379/0
-- *** --- * --- .> concurrency: 12 (prefork)
-- ***** ----- .> task events: OFF (enable -E to monitor tasks in this worker)
-- ***** -----
-- ***** ----- [queues]
-- ***** ----- .> celery exchange=celery(direct) key=celery

[tasks]
. tasks.square_root

[2021-11-28 00:42:37,352: INFO/MainProcess] Connected to redis://localhost:6379/0
[2021-11-28 00:42:37,363: INFO/MainProcess] mingle: searching for neighbors
[2021-11-28 00:42:38,383: INFO/MainProcess] mingle: all alone
[2021-11-28 00:42:38,402: INFO/MainProcess] celery@LAPTOP-C8GE10C8 ready.
[]
```

Figure 5 celery worker is running and connected to redis at localhost:6379 which is running in a docker container

```
~/projects/dsf_assignemnt
env > python3 publisher.py 15
square root of 15 is 3.872983346207417

~/projects/dsf_assignemnt
env > python3 publisher.py 16
square root of 16 is 4.0

~/projects/dsf_assignemnt
env > python3 publisher.py 25
square root of 25 is 5.0

~/projects/dsf_assignemnt
env > []
```

Figure 6 trigger from publisher.py

```
~/projects/dsf_assignemnt
env > celery -A tasks worker --loglevel=INFO

----- celery@LAPTOP-C8GE10C8 v5.2.1 (dawn-chorus)
-- ***** -----
-- ***** ----- Linux-5.10.60.1-microsoft-standard-WSL2-x86_64-with-glibc2.29 2021-11-28 00:42:37
-- *** --- * ---
-- ** ----- [config]
-- ** ----- .> app: tasks:0x7f4a5b031df0
-- ** ----- .> transport: redis://localhost:6379/0
-- ** ----- .> results: redis://localhost:6379/0
-- *** --- * --- .> concurrency: 12 (prefork)
-- ***** ----- .> task events: OFF (enable -E to monitor tasks in this worker)
-- ***** -----
-- ***** ----- [queues]
-- ***** ----- .> celery exchange=celery(direct) key=celery

[tasks]
. tasks.square_root

[2021-11-28 00:42:37,352: INFO/MainProcess] Connected to redis://localhost:6379/0
[2021-11-28 00:42:37,363: INFO/MainProcess] mingle: searching for neighbors
[2021-11-28 00:42:38,383: INFO/MainProcess] mingle: all alone
[2021-11-28 00:42:38,402: INFO/MainProcess] celery@LAPTOP-C8GE10C8 ready.
[2021-11-28 00:43:37,407: INFO/MainProcess] Task tasks.square_root[e9ecd58a-c70b-43eb-b499-371778d0513c] received
[2021-11-28 00:43:37,413: INFO/ForkPoolWorker-7] Task tasks.square_root[e9ecd58a-c70b-43eb-b499-371778d0513c] succeeded in 0.005017099996621255s: 'square root of 15 is 3.872983346207417'
[2021-11-28 00:43:44,888: INFO/MainProcess] Task tasks.square_root[cd5b42e7-f2a8-4986-86d5-531e40625531] received
[2021-11-28 00:43:44,891: INFO/ForkPoolWorker-7] Task tasks.square_root[cd5b42e7-f2a8-4986-86d5-531e40625531] succeeded in 0.002121500001521781s: 'square root of 16 is 4.0'
[2021-11-28 00:43:48,290: INFO/MainProcess] Task tasks.square_root[55705b57-9bab-47cd-87cd-46e738cd558e] received
[2021-11-28 00:43:48,293: INFO/ForkPoolWorker-7] Task tasks.square_root[55705b57-9bab-47cd-87cd-46e738cd558e] succeeded in 0.00238760000502225s: 'square root of 25 is 5.0'
[]
```

Figure 7 we can see the tasks captured in celery worker

```
docker exec -it 7273b87c50b0: x + v
#
# redis-cli
127.0.0.1:6379>
127.0.0.1:6379> keys *
1) "celery-task-meta-bf320cc4-71d0-400e-bcc8-55d44f31afa1"
2) "celery-task-meta-844ecd6d-5a9b-4a14-a504-915e026fb485"
3) "celery-task-meta-55705b57-9bab-47cd-87cd-46e738cd558e"
4) "celery-task-meta-6ff83cde-25d4-4309-982b-805573af1c8a"
5) "_kombu.binding.celery.pidbox"
6) "celery-task-meta-cd5b42e7-f2a8-4986-86d5-531e40625531"
7) "_kombu.binding.celery"
8) "_kombu.binding.celeryev"
9) "celery-task-meta-d54d75ae-ffcd-495f-a059-6453bed7304c"
10) "celery-task-meta-e9ecd58a-c70b-43eb-b499-371778d0513c"
11) "celery-task-meta-ae0f93c5-8509-4cad-964e-ac1ed6ce9f55"
12) "celery-task-meta-8fafe02a-f480-48db-b529-84ea8f97ec60"
127.0.0.1:6379>
127.0.0.1:6379> get celery-task-meta-d54d75ae-ffcd-495f-a059-6453bed7304c
"{\"status\": \"SUCCESS\", \"result\": \"square root of 25 is 5.0\", \"traceback\": null, \"children\": [], \"date_done\": \"2021-11-27T19:15:16.774029\", \"task_id\": \"d54d75ae-ffcd-495f-a059-6453bed7304c\"}"
127.0.0.1:6379>
127.0.0.1:6379> get celery-task-meta-8fafe02a-f480-48db-b529-84ea8f97ec60
"{\"status\": \"SUCCESS\", \"result\": \"square root of 15 is 3.872983346207417\", \"traceback\": null, \"children\": [], \"date_done\": \"2021-11-27T19:15:04.496228\", \"task_id\": \"8fafe02a-f480-48db-b529-84ea8f97ec60\"}"
127.0.0.1:6379>
```

Figure 8 we can check the celery meta data stored in redis database

1. <https://docs.celeryproject.org/en/stable/getting-started/introduction.html>
2. <https://docs.python.org/3/library/venv.html>
3. <https://pandas.pydata.org/docs/reference/io.html>
4. https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html