Part 1

1. You have a Python script that processes millions of records in a single thread. How would you optimize it to leverage multiple cores and reduce the execution time? Provide a sample code snippet.

To leverage multiple cores we have 2 choices. First of all, we can use `multiprocessing` module:

```python
import multiprocessing

# Define the function to process each chunk
def single_process(args)

# Define the function to split the records into exactly n_chunks
def split_records(args)

def parallel_processing(single_process_function, records, pool_size):

    record_chunks = split_records(records, pool_size)
    final_results = None

    with multiprocessing.Pool(pool_size) as pool:
        results = pool.map_async(single_process_function, record_chunks)
        final_results = results.get()
    return final_results

if __name__ == "__main__":
    records # Assuming records are available
    pool_size # Adjust pool size based on available CPU cores
    processed_records = parallel_processing(single_process, records, pool_size)

    # Save data
```

A custom split_records funtion is defined as follows:

```python
def split_records(records, n_chunks):
    chunk_size = len(records) // n_chunks
    chunks = [records[i:i + chunk_size] for i in range(0, n_chunks-1)]
    chunks.append(records[n_chunks*chunk_size-chunk_size:])
    return chunks
```

Sometimes there is a built-in functionality to split data into chunks. For example numpy provide the `array_split` funtion and map_async has its own parameter to handle such cases called `chunksize`. Of course instead of map_appy we could use any other method provided by multiprocessing module.

The second option is to use spark through pyspark. In this way we can benefit from a whole cluster and hence a large number of processors. Below, it is a snippet applying a transformation called `single_process_udf` defined from a custom transformation `single_process` in order to be used by spark:

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import *

spark = SparkSession.builder.appName("mymain").getOrCreate()

# read data from a warehouse/file/database/etc
df = spark.read.format(data_format).option(option,
option_value).schema(data_schema).load(path_to_data)

# Define the transformation
def single_process(*cols)

single_process_udf = udf(single_process, Appropriate_Type())

# Apply the transformation
result_df = df.withColumn("result", custom_udf(cols_to_consinder))

# save data
```

Finally there many other libraries like Dask or Ray that assist in parallel computing with their own pros and cons.

2. During a data pipeline run in Azure Data Factory, a step failed due to an invalid data format. Describe how you would debug this issue and prevent it from happening in the future.

In such a scenario, we would start by using the Azure Data Factory Monitoring tools to identify which pipeline failed. The next step is to check the logs and error messages to determine the specific cause of the failure, such as which column or data type caused the issue. Additionally, we can use the Debug feature in ADF to walk through the pipeline step by step and inspect the data at each stage.

To prevent such errors from happening we can take the following measures:

- Preview the Dataset: To prevent these errors in the future firstly we need to perform a quick preview of the dataset to ensure that the schema and data types match the expected format.
- Data Validation: We could implement a schema validation step early in the pipeline to detect and flag any data type mismatches. This can help to spot errors earlier and stop the pipeline before it reaches critical stages.
- Alert System: Finally we can set up an alert system in order to send a message when an error occurs, allowing for a faster response.

In terms of handling errors or exceptions, there are several approaches based on the severity, frequency and fault tolerance of the issue:

- Immediate Failure for Critical Errors: If the error is rare but critical, an instant failure is recommended to force a review and correction of the invalid data. This approach ensures all data is processed correctly. Sometimes may require manual intervention for a few exceptional records.
- Type Casting for Consistent Errors: If the error occurs more frequently and follows a predictable pattern, it's better to add a data transformation or cast the type to the correct format within the pipeline. This can resolve common discrepancies without requiring manual intervention, making the pipeline more resilient.
- Graceful Failure Handling: For non-critical errors, we can implement an error-handling process. For example, set up failure paths where specific actions are taken when a failure occurs (e.g., logging the error and proceeding with the rest of the data). This ensures that the pipeline does not stop entirely, while still handling problematic records following the "on failure" path.
- Two Data Layers with Data Cleansing Pipeline: If a large portion of the incoming data is inconsistent, we can create a separate pipeline dedicated to data cleansing and preprocessing. This pipeline will handle data correction, validation, and transformation before the data is passed to the main processing pipeline. Many organizations maintain two data layers or pools. A raw data layer that stores unprocessed data exactly as received from external sources and a cleansed data layer that contains validated and corrected data, ready for processing by pipelines.

3. Write a Python script using the Azure SDK that uploads a file to an Azure Blob Storage container. Ensure the script checks if the container exists and creates it if it does not.

This is a simple snippet which connects with a container if exists otherwise it creates it and then try to upload the blob.

```python
from azure.storage.blob import BlobServiceClient, BlobClient, ContainerClient

# Assuming below entities are available
connection_string # connection string (simple connection)
container_name # container name
blob_name # blob name
file_path # Assuming the file exists under file_path

blob_service_client = BlobServiceClient.from_connection_string(connection_string)

container_client = blob_service_client.get_container_client(container_name)

# Check if the container exists, if not, create it
try:
    if not container_client.exists():
        container_client.create_container()
    else:
        print(f"Container '{container_name}' already exists.")
except Exception as e:
    print("An error occured:")
    print(f"Error message: {str(e)}")

blob_client = container_client.get_blob_client(blob_name)

try:
    # overwrite True if the blob already exists
    with open(file_path, "rb") as data:
        blob_client.upload_blob(data, overwrite=True)
except Exception as e:
    print("An error occured:")
    print(f"Error message: {str(e)}")
```

We could handle the errors in a better way (eg. logging, retry policies etc) but it is out of scope so we just used the classic try-except clause. Further more there are other options to connect with azure (eg. using Azure AD, Managed Identity, RBAC, Tokens) but we kept things simple with the connection_string.

4. Write a Python script to download logs from Azure (e.g. events from a specific resource)

In the following snippet we retrieve logs from `table` (table is a parameter for example it could be syslog table). The script retrieves the logs for a period of 24 hours (notice: datetime.now() - timedelta(days=1), datetime.now()).

```
from azure.identity import ClientSecretCredential
from azure.monitor.query import LogsQueryClient
from datetime import datetime, timedelta

# Assuming everything is available
tenant_id
client_id
client_secret
log_analytics_workspace_id
vm_resource_id
table

credential = ClientSecretCredential(
    tenant_id=tenant_id,
    client_id=client_id,
    client_secret=client_secret
)

client = LogsQueryClient(credential)

query = f"""
{table}
| where ResourceId == "{vm_resource_id}"
"""

response = client.query_workspace(
    workspace_id=log_analytics_workspace_id,
    query=query,
    timespan=(datetime.now() - timedelta(days=1), datetime.now())
)

if response.status == "Success":
    # save results
    pass
```

```
else:
    print(f"Query failed: {response.error}")
```