

App Dev Midterm Cheat Sheet

1. Python Basics

JSON Handling

Importing and Loading JSON Data

```
import json

# JSON string
json_data = '{"name": "Alice", "age": 25, "city": "New York"}'

# Convert JSON string to dictionary
data = json.loads(json_data)
```

Writing and Reading JSON Files

```
# Writing JSON to a file
with open('data.json', 'w') as f:
    json.dump(data, f, indent=4)

# Reading JSON from a file
with open('data.json', 'r') as f:
    loaded_data = json.load(f)
print(loaded_data)
```

Requesting APIs

Using the `requests` Library

```
import requests

# Making a GET request
response = requests.get('https://test.com')

# Converting response to JSON
data = response.json()
```

Handling API Responses

```
if response.status_code == 200:
    print("Request was successful!")
```

```
else:
    print(f"Error: {response.status_code}")
```

Sending Data with POST Request

```
payload = {"name": "Alice", "age": 25}
headers = {'Content-Type': 'application/json'}
response = requests.post('https://test.com/posts', json=payload, headers=headers)
print(response.json())
```

List Comprehension

Basic List Comprehension

```
numbers = [1, 2, 3, 4, 5]
squared = [x**2 for x in numbers]
print(squared) # Output: [1, 4, 9, 16, 25]
```

Filtering with List Comprehension

```
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers) # Output: [2, 4]
```

2. Pandas

Create DataFrame

```
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
```

Read CSV

```
df = pd.read_csv('data.csv')
```

Basic DataFrame Operations

```
# Display First Few Rows
df.head()
```

```

df.head(10)

# Display Last Few Rows
df.tail()

# Check Data Types
df.dtypes

# Get Basic Info
df.info()

# Get Summary Statistics
df.describe()

# Value Counts (Quick Category Summary)
df['Category'].value_counts()

```

Indexing

```

# Select
df.loc[0], df.iloc[0]      # First row (label vs position)
df.loc[:, 'Age'], df.iloc[:, 1] # 'Age' column (label vs position)

# Filter
df.loc[df['Age'] > 25]      # By condition
df.iloc[0:2]               # First two rows

# Update
df.loc[df['Name'] == 'Alice', 'Age'] = 26
df.iloc[0, 1] = 27

# Add Row
df.loc[len(df)] = ['Charlie', 35]

```

- `.loc` selects data by label (index/column name), while `.iloc` selects data by position (integer index)

Pivot Tables in Pandas

Creating a Pivot Table

```

import pandas as pd

# Sample Data
data = {
    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
    'SubCategory': ['X', 'X', 'Y', 'Y', 'X', 'Y'],
    'Sales': [100, 200, 150, 300, 250, 400]
}

df = pd.DataFrame(data)

```

```
# Creating a Pivot Table
pivot_table = df.pivot_table(values='Sales', index='Category',
                              columns='SubCategory', aggfunc='sum', fill_value=0)

print(pivot_table)
```

Explanation

- `values='Sales'`: Specifies which column's values to aggregate.
- `index='Category'`: Rows are grouped by "Category."
- `columns='SubCategory'`: Creates columns based on "SubCategory."
- `aggfunc='sum'`: Aggregates using sum (you can use `mean`, `count`, etc.).
- `fill_value=0`: Replaces NaN values with 0.

Additional Aggregations

```
# Using Multiple Aggregations
pivot_table_multi = df.pivot_table(values='Sales', index='Category',
                                    columns='SubCategory', aggfunc=['sum', 'mean'], fill_value=0)

print(pivot_table_multi)
```

Resetting Index

```
pivot_table_reset = pivot_table.reset_index()
print(pivot_table_reset)
```

Selecting and Filtering Data

```
# Selecting a Single Column
df['Age']

# Selecting Multiple Columns
df[['Name', 'Age']]

# Filtering Rows Based on Condition
df_filtered = df[df['Age'] > 25]

# Filtering with Multiple Conditions
df_filtered = df[(df['Age'] > 25) & (df['Name'] == 'Alice')]
```

Modifying Data

```
# Adding a New Column
df['Salary'] = [50000, 60000]

# Updating Values in a Column
df.loc[df['Name'] == 'Alice', 'Age'] = 26

# Dropping a Column
df.drop(columns=['Salary'], inplace=True)
```

Sorting and Grouping Data

```
# Sorting Data by Column
df_sorted = df.sort_values(by='Age', ascending=False)

# Grouping and Aggregating Data
df.groupby('Category').mean()
```

Handling Missing Data

```
# Checking for Missing Values
df.isnull().sum()

# Dropping Rows with Missing Values
df.dropna(inplace=True)

# Filling Missing Values
df.fillna(value=0, inplace=True)
```

Merging and Joining DataFrames

```
# Concatenating DataFrames
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
df_concat = pd.concat([df1, df2])

# Merging DataFrames on a Column
df1 = pd.DataFrame({'ID': [1, 2], 'Name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'ID': [1, 2], 'Age': [25, 30]})
df_merged = pd.merge(df1, df2, on='ID')
```

3. Matplotlib

Basics

Line Plot

```
import matplotlib.pyplot as plt

plt.plot(x, y)
plt.show()
```

- Available chart options: `scatter`, `bar`, `hist`, `boxplot`, etc.

Bar Chart

```
plt.bar(categories, values)
plt.show()
```

Histogram

```
plt.hist(data, bins=10)
plt.show()
```

Customisation

Adjusting Figure Size

```
plt.figure(figsize=(10, 6))
```

Customizing Colors

```
plt.plot(x, y, color='red')
```

- Available colors: `b` (blue), `g` (green), `r` (red), `c` (cyan), `m` (magenta), `y` (yellow), `k` (black), `w` (white).

Modifying Axis Labels and Titles

```
plt.xlabel('X-Axis Label')
plt.ylabel('Y-Axis Label')
plt.title('Plot Title')
```

Adding Grid Lines

```
plt.grid(True)
```

Adding Legends

```
plt.legend(['Series 1', 'Series 2'])
```

4. Seaborn

Basics

Scatter Plot

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.scatterplot(x='Age', y='Salary', data=df)
plt.show()
```

- Available chart options: `histplot`, `boxplot`, `lineplot`, `barplot`, etc.

Heatmap

```
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
```

Customisation

Changing the Theme

```
sns.set_theme(style='darkgrid')
```

- Available themes: `darkgrid`, `whitegrid`, `dark`, `white`, `ticks`.

Adjusting Figure Size

```
plt.figure(figsize=(10, 6))
```

Customizing Colors

```
sns.set_palette('pastel')
```

- Available palettes: `deep`, `muted`, `bright`, `pastel`, `dark`, `colorblind`.

Modifying Axis Labels and Titles

```
plt.xlabel('X-Axis Label')
plt.ylabel('Y-Axis Label')
plt.title('Plot Title')
```

Adding Grid Lines

```
plt.grid(True)
```

5. Plotly

Basics

Scatter Plot

```
import plotly.express as px
fig = px.scatter(df, x='Age', y='Salary')
fig.show()
```

- Available chart options: `histogram`, `box`, `line`, `bar`, etc.

Heatmap

```
import plotly.figure_factory as ff
import numpy as np

corr_matrix = df.corr().to_numpy()
fig = ff.create_annotated_heatmap(z=corr_matrix,
                                x=df.columns.tolist(),
                                y=df.columns.tolist(),
                                colorscale='coolwarm')
```

Customisation

Adjusting Figure Size

```
fig.update_layout(width=800, height=500)
```


Customizing Colors

```
fig.update_traces(marker=dict(color='lightblue'))
```

- Available colours: **Viridis**, **Cividis**, **Plasma**, etc.

Modifying Axis Labels and Titles

```
fig.update_layout(  
    title='Plot Title',  
    xaxis_title='X-Axis Label',  
    yaxis_title='Y-Axis Label'  
)
```

Adding Grid Lines

```
fig.update_xaxes(showgrid=True)  
fig.update_yaxes(showgrid=True)
```

6. MongoDB

Connect to MongoDB

```
from pymongo import MongoClient  
client = MongoClient('mongodb://localhost:27017/') # Connect to MongoDB  
db = client['mydatabase'] # Select database  
collection = db['users'] # Select collection
```

Insert Document

```
# Insert a single document:  
collection.insert_one({'name': 'Alice', 'age': 25})  
  
# Insert multiple documents:  
users = [  
    {'name': 'Bob', 'age': 30},  
    {'name': 'Charlie', 'age': 28}  
]  
collection.insert_many(users)
```

Find Document

```
# Find one document:
user = collection.find_one({'name': 'Alice'})
print(user)

# Find all documents:
for user in collection.find():
    print(user)

# Find documents with a condition:
for user in collection.find({'age': {'$gt': 25}}):
    print(user)
```

Update Document

```
# Update one document:
collection.update_one({'name': 'Alice'}, {'$set': {'age': 26}})

# Update multiple documents:
collection.update_many({'age': {'$lt': 30}}, {'$set': {'status': 'young'}})
```

Delete Document

```
# Delete one document:
collection.delete_one({'name': 'Alice'})

# Delete multiple documents:
collection.delete_many({'age': {'$gt': 30}})
```

Filtering and Query Operators

Operator	Description	Example
<code>\$gt</code>	Greater than	<code>{'age': {'\$gt': 25}}</code>
<code>\$lt</code>	Less than	<code>{'age': {'\$lt': 30}}</code>
<code>\$gte</code>	Greater than or equal	<code>{'age': {'\$gte': 18}}</code>
<code>\$lte</code>	Less than or equal	<code>{'age': {'\$lte': 50}}</code>
<code>\$ne</code>	Not equal	<code>{'age': {'\$ne': 30}}</code>
<code>\$in</code>	Matches any value in list	<code>{'name': {'\$in': ['Alice', 'Bob']}}</code>
<code>\$exists</code>	Checks if a field exists	<code>{'status': {'\$exists': True}}</code>

7. MySQL

Connect to MySQL

```
import mysql.connector
conn = mysql.connector.connect(host='localhost', user='root', password='',
database='mydb')
cursor = conn.cursor()
```

Create Database

```
cursor.execute("CREATE DATABASE mydb")
cursor.execute("USE mydb")
```

Create Table

```
cursor.execute("""
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    age INT
)
""")
```

Insert Data

```
# Insert a single record:
cursor.execute("INSERT INTO users (name, age) VALUES (%s, %s)", ('Alice', 25))

# Insert multiple records:
users = [
    ('Bob', 30),
    ('Charlie', 28)
]
cursor.executemany("INSERT INTO users (name, age) VALUES (%s, %s)", users)
conn.commit()
```

Select Data

```
# Retrieve all records:
cursor.execute("SELECT * FROM users")

# Retrieve specific columns:
cursor.execute("SELECT name FROM users")

# Filter data using WHERE:
cursor.execute("SELECT * FROM users WHERE age > 25")
```

```
for row in cursor.fetchall():
    print(row)
```

Update Data

```
cursor.execute("UPDATE users SET age = 26 WHERE name = 'Alice'")
conn.commit()
```

Delete Data

```
# Delete a specific record:
cursor.execute("DELETE FROM users WHERE name = 'Alice'")

# Delete all records (Be careful!!!):
cursor.execute("DELETE FROM users")
conn.commit()
```

SQL Query Operators

Operator	Description	Example
=	Equal to	SELECT * FROM users WHERE age = 25
!= or <>	Not equal	SELECT * FROM users WHERE age <> 30
>	Greater than	SELECT * FROM users WHERE age > 25
<	Less than	SELECT * FROM users WHERE age < 30
>=	Greater than or equal	SELECT * FROM users WHERE age >= 18
<=	Less than or equal	SELECT * FROM users WHERE age <= 50
BETWEEN	Between two values	SELECT * FROM users WHERE age BETWEEN 20 AND 30
IN	Matches any value in list	SELECT * FROM users WHERE name IN ('Alice', 'Bob')
LIKE	Pattern matching	SELECT * FROM users WHERE name LIKE 'A%'

MySQL Joins

INNER JOIN (Returns matching records in both tables)

```
SELECT users.name, orders.product
FROM users
INNER JOIN orders ON users.id = orders.user_id;
```

LEFT JOIN (Returns all records from left table and matching from right)

```
SELECT users.name, orders.product
FROM users
LEFT JOIN orders ON users.id = orders.user_id;
```

RIGHT JOIN (Returns all records from right table and matching from left)

```
SELECT users.name, orders.product
FROM users
RIGHT JOIN orders ON users.id = orders.user_id;
```

FULL JOIN (MySQL doesn't support it directly, but can be simulated with UNION)

```
SELECT users.name, orders.product
FROM users
LEFT JOIN orders ON users.id = orders.user_id
UNION
SELECT users.name, orders.product
FROM users
RIGHT JOIN orders ON users.id = orders.user_id;
```

8. Firebase

Initialize Firebase

```
import firebase_admin
from firebase_admin import credentials, firestore
cred = credentials.Certificate('path/to/your/serviceAccountKey.json')
firebase_admin.initialize_app(cred)
db = firestore.client()
```

Add Data

```
# Add Data (Create Record):
db.collection('users').add({'name': 'Alice', 'age': 25})

# To use a specific document ID:
db.collection('users').document('alice_id').set({'name': 'Alice', 'age': 25})
```

Get Data

```
# Retrieve all documents in a collection:
docs = db.collection('users').get()
for doc in docs:
    print(doc.to_dict())

# Retrieve a specific document by ID:
user_ref = db.collection('users').document('alice_id')
doc = user_ref.get()
if doc.exists:
    print(doc.to_dict())
else:
    print("No such document!")

# Query based on conditions:
users = db.collection('users').where('age', '>=', 25).get()
for user in users:
    print(user.to_dict())
```

Update Data

```
user_ref = db.collection('users').document('user_id')
user_ref.update({'age': 26})
```

Delete Data

```
# Delete a document:
db.collection('users').document('user_id').delete()

# Delete a field from a document:
user_ref = db.collection('users').document('alice_id')
user_ref.update({'age': firestore.DELETE_FIELD})
```

Firestore Query Operators

Operator	Description	Example
<code>==</code>	Equal to	<code>where('age', '==', 25)</code>
<code>!=</code>	Not equal	<code>where('age', '!=', 30)</code>
<code>></code>	Greater than	<code>where('age', '>', 25)</code>
<code><</code>	Less than	<code>where('age', '<', 30)</code>
<code>>=</code>	Greater than or equal	<code>where('age', '>=', 18)</code>
<code><=</code>	Less than or equal	<code>where('age', '<=', 50)</code>
<code>in</code>	Matches any value in list	<code>where('name', 'in', ['Alice', 'Bob'])</code>

Operator	Description	Example
array-contains	Checks if an array contains a value	<code>where('hobbies', 'array-contains', 'reading')</code>