SET-I

1. a) Naming Rules for Python Identifiers

In Python, identifiers are names used to identify variables, functions, classes, modules, and other objects. The rules for naming identifiers are as follows:

1. **Alphabetic Characters and Digits**: Identifiers can include both alphabetic characters (A-Z, a-z) and digits (0-9). For example, variable1 and data2 are valid identifiers.
2. **Underscore**: The underscore (_) character is allowed and often used to separate words in an identifier. For instance, my_variable and data_set are valid identifiers.
3. **No Starting with Digits**: Identifiers cannot start with a digit. For example, 123variable is invalid, but variable123 is valid.
4. **Case Sensitivity**: Identifiers are case-sensitive. For instance, Variable and variable are considered different identifiers.
5. **Keywords Restriction**: Identifiers cannot be Python keywords (reserved words). For example, class, for, and if cannot be used as identifiers.
6. **Special Characters**: Identifiers cannot include special characters like @, #, !, etc.

Python's naming rules ensure that identifiers are meaningful and avoid conflicts with reserved keywords. For example, using my_variable instead of var makes the code more readable and understandable.

1. b) Lambda Function

A lambda function in Python is a small anonymous function defined using the lambda keyword. It can have any number of arguments but only one expression. The syntax is lambda arguments: expression. Lambda functions are used for creating small, throwaway functions without needing to formally define them using the def keyword. They are often used in situations where a simple function is required for a short period, such as in map(), filter(), and sorted() functions. For example:

```
# Using lambda function with map

numbers = [1, 2, 3, 4]

squared = list(map(lambda x: x**2, numbers))

print(squared)  # Output: [1, 4, 9, 16]
```

Lambda functions are particularly useful in functional programming paradigms where functions are passed as arguments to other functions. They provide a concise way to define simple functions inline without cluttering the code with formal function definitions.

2. a) Else Statement with For and While Loop

The else statement in Python can be used with both for and while loops. It is executed when the loop completes normally, i.e., without encountering a break statement. For example:

```
for i in range(5):
    if i == 3:
        break
else:
    print("Loop completed without break")
```

In this example, the else block will not execute because the loop is terminated by the break statement. However, if the loop completes without a break, the else block will execute. This feature is useful for scenarios where you need to perform an action only if the loop wasn't interrupted by a break.

Similarly, the else statement can be used with a while loop:

```
i = 0
while i < 5:
    if i == 3:
        break
    i += 1
else:
    print("Loop completed without break")
```

In this case, the else block will not execute because the loop is terminated by the break statement when i equals 3.

2. b) String Functions

- **upper()**: Converts all characters in a string to uppercase.
  Example: "hello".upper() returns "HELLO".
- **lower()**: Converts all characters in a string to lowercase.
  Example: "HELLO".lower() returns "hello".
- **isdigit()**: Checks if all characters in a string are digits.
  Example: "123".isdigit() returns True.
- **isalpha()**: Checks if all characters in a string are alphabetic.
  Example: "abc".isalpha() returns True.
- **split()**: Splits a string into a list of substrings based on a delimiter.
  Example: "a,b,c".split(",") returns ['a', 'b', 'c'].

- **join()**: Joins a list of strings into a single string with a specified delimiter. Example: ",".join(['a', 'b', 'c']) returns "a,b,c".

These string functions are essential for manipulating and processing text data in Python. They provide a straightforward way to perform common string operations, making it easier to handle and transform text.

3. a) Modes for Reading and Writing Data into a File

Python provides several modes for reading and writing files:

- **'r'**: Read mode (default). Opens a file for reading.
- **'w'**: Write mode. Opens a file for writing (creates a new file or truncates an existing file).
- **'a'**: Append mode. Opens a file for appending (creates a new file if it doesn't exist).
- **'r+'**: Read and write mode. Opens a file for both reading and writing.
- **'b'**: Binary mode. Used with other modes to read/write binary files.

For example, to write to a file in write mode:

```
with open('example.txt', 'w') as file:
    file.write("Hello, World!")
```

And to read from a file in read mode:

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

These modes provide flexibility in handling files, allowing you to choose the appropriate mode based on your requirements.

3. b) Differences Between List, Tuple, Set, and Dictionary

- **List**: Ordered, mutable collection of items. Example: [1, 2, 3]. Lists allow duplicate elements and can be modified after creation.
- **Tuple**: Ordered, immutable collection of items. Example: (1, 2, 3). Tuples are similar to lists but cannot be modified after creation.
- **Set**: Unordered, mutable collection of unique items. Example: {1, 2, 3}. Sets do not allow duplicate elements and are used for membership testing and eliminating duplicates.
- **Dictionary**: Unordered, mutable collection of key-value pairs. Example: {'a': 1, 'b': 2}. Dictionaries allow fast retrieval of values based on keys.

These data structures provide different ways to store and manipulate data, each with its own advantages and use cases. Lists and tuples are suitable for ordered collections, sets are ideal for unique items, and dictionaries are perfect for key-value pairs.

SET-II

4. a) User-Defined Exceptions

User-defined exceptions in Python can be created by defining a new class that inherits from the built-in Exception class. Example:

```
class CustomError(Exception):
    pass


try:
    raise CustomError("An error occurred")
except CustomError as e:
    print(e)
```

In this example, CustomError is a user-defined exception that can be raised and caught like any other exception. This allows you to create specific exceptions for your application, providing more meaningful error handling.

4. b) Regular Expression for Validating Indian Mobile Numbers

A regular expression for validating Indian mobile numbers (format: +91-XXXXXXXXXX) is:

```
import re


pattern = r'^\+91-\d{10}$'
number = "+91-9876543210"
if re.match(pattern, number):
    print("Valid number")
else:
    print("Invalid number")
```

This regular expression ensures that the number starts with +91- followed by exactly 10 digits. Regular expressions are powerful tools for pattern matching and validation, making them useful for tasks like input validation.

5. a) Game Loops and Event Handling

Game loops are the core of any game, responsible for updating the game state and rendering graphics. Event handling manages user inputs and other events. Together, they ensure the game runs smoothly and responds to user actions. For example, in a simple game loop:

```
import pygame

pygame.init()
screen = pygame.display.set_mode((800, 600))
running = True

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    # Update game state
    # Render graphics
    pygame.display.flip()

pygame.quit()
```

In this example, the game loop continuously checks for events (such as user inputs) and updates the game state accordingly. Event handling ensures that the game responds to user actions, such as key presses or mouse clicks.

5. b) Data Preprocessing Steps

Data preprocessing involves several steps: data cleaning (handling missing values, outliers), data transformation (normalization, encoding), and data reduction (feature selection, dimensionality reduction). For example, handling missing values:

```
import pandas as pd

data = pd.read_csv('data.csv')
data.fillna(data.mean(), inplace=True)
```

In this example, missing values in the dataset are replaced with the mean of the respective columns. Data preprocessing is a crucial step in data analysis and machine learning, as it ensures that the data is clean and suitable for modeling.

6. a) PEP8 Guidelines

PEP8 is the style guide for Python code, emphasizing readability and consistency. Key guidelines include:

1. **Indentation**: Use 4 spaces per indentation level. Avoid using tabs.
2. def example_function():
3. print("Hello, World!")
4. **Maximum Line Length**: Limit all lines to a maximum of 79 characters. For longer lines, use parentheses for implicit line continuation.
5. long_string = ("This is a very long string that needs to be split "
6. "into multiple lines for better readability.")
7. **Blank Lines**: Use blank lines to separate top-level function and class definitions, and within functions to separate logical sections.
8. class MyClass:
9. def __init__(self):
10. self.value = 0
11.
12. def increment(self):
13. self.value += 1
14. **Imports**: Imports should usually be on separate lines and at the top of the file. Group imports in the following order: standard library imports, related third-party imports, and local application/library-specific imports.
15. import os
16. import sys
17.
18. import numpy as np
19. import pandas as pd
20.
21. from mymodule import myfunction
22. **Whitespace in Expressions and Statements**: Avoid extraneous whitespace in the following situations:
    - Immediately inside parentheses, brackets, or braces.

- Before a comma, semicolon, or colon.
- Before the open parenthesis that starts the argument list of a function call.
- Around operators with the lowest priority.

```
# Correct:
spam(ham[1], {eggs: 2})
x = 1
y = x + 1

# Incorrect:
spam( ham[ 1 ], { eggs: 2 } )
x  =  1
y  =  x  +  1
```

Naming Conventions: Follow naming conventions for different types of identifiers:

- Function names should be lowercase, with words separated by underscores.
- Variable names should be lowercase, with words separated by underscores.
- Class names should use the CapWords convention.
- Constants should be written in all capital letters with underscores separating words.

```
def my_function():
    pass

my_variable = 10

class MyClass:
    pass

MAX_VALUE = 100
```

Comments: Use comments to explain code. Comments should be complete sentences and start with a capital letter. Use inline comments sparingly.

```
# This is a comment explaining the following line of code
```

44. x = x + 1 # Increment x by 1
45. **Docstrings**: Use docstrings to describe all public modules, functions, classes, and methods. Docstrings should be enclosed in triple quotes.
46. def my_function():
47. """
48. This is a docstring explaining what the function does.
49. """
50. pass

Following PEP8 guidelines helps maintain a consistent coding style, making the code more readable and maintainable.

6. b) CRUD Operations

CRUD stands for Create, Read, Update, and Delete. These are the four basic operations for managing data in databases. Each operation corresponds to a specific SQL command:

1. **Create**: Adds new records to a database table. The SQL command for this operation is INSERT.
2. INSERT INTO users (name, age) VALUES ('Alice', 30);
3. **Read**: Retrieves data from a database table. The SQL command for this operation is SELECT.
4. SELECT * FROM users;
5. **Update**: Modifies existing records in a database table. The SQL command for this operation is UPDATE.
6. UPDATE users SET age = 31 WHERE name = 'Alice';
7. **Delete**: Removes records from a database table. The SQL command for this operation is DELETE.
8. DELETE FROM users WHERE name = 'Alice';

CRUD operations are fundamental to database management, allowing for the creation, retrieval, modification, and deletion of data. These operations are essential for maintaining and manipulating data in any database-driven application.