second_week

January 15, 2025

1 CS2201

2 Week 2

2.1 Set 2

2.1.1 Solution by Abhisek Sarkar

as20ms091@iiserkol.ac.in

2.1.2 Extra Resources:

- For patterns in python look: https://www.geeksforgeeks.org/programs-printing-pyramid-patterns-python/

2.1.3 Q1.

Write a program which prints the following sequence of strings using a for loop

```
mitochondria
mitochondria
mitochondri
mitochondr
mitochond
mitochon
mitocho
mitoch
mitoc
mito
mit
mi
m
```

[1]:
```python
def print_sequence():
  """Prints the given sequence of strings using a for loop."""
  word = "mitochondria"
  for i in range(len(word), 0, -1):
    print(word[:i])

# Call the function to print the sequence
```

```
print_sequence()
```

```
mitochondria
mitochondri
mitochondr
mitochond
mitochon
mitocho
mitoch
mitoc
mito
mit
mi
m
```

Explanation:

1. `def print_sequence():`
   - This line defines a function named `print_sequence` that will handle the printing of the sequence.
2. `word = "mitochondria"`
   - This line assigns the string "mitochondria" to the variable `word`.
3. `for i in range(len(word), 0, -1):`
   - This is the main loop.
     - `range(len(word), 0, -1)` creates a sequence of numbers starting from the length of the word (`len(word)`) and decrementing by 1 in each iteration until it reaches 1. This ensures that the loop iterates through the desired number of characters.
4. `print(word[:i])`
   - This line prints a slice of the original word.
     - `word[:i]` extracts a substring from the beginning of the word (`word`) up to the index `i-1`. This effectively shortens the word in each iteration, producing the desired sequence.
5. `print_sequence()`
   - This line calls the `print_sequence` function to execute the code within it and print the sequence to the console.

This code will output the following:

```
mitochondria
mitochondria
mitochondri
mitochondr
mitochond
mitochon
mitocho
mitoch
mitoc
mito
mit
mi
```

m

2.1.4 Q2.

Consider a list of names of chemical elements (e.g. sodium, potassium etc.) and write a program that finds the longest name in that list. Using "format()" print the longest name thus found with its length.

[2]:
```
elements_list = [
    "Hydrogen",
    "Helium",
    "Lithium",
    "Beryllium",
    "Boron",
    "Carbon",
    "Nitrogen",
    "Oxygen",
    "Fluorine",
    "Neon",
    "Sodium",
    "Magnesium",
    "Aluminum",
    "Silicon",
    "Phosphorus",
    "Sulfur",
    "Chlorine",
    "Argon",
    "Potassium",
    "Calcium",
    "Scandium",
    "Titanium",
    "Vanadium",
    "Chromium",
    "Manganese",
    "Iron",
    "Cobalt",
    "Nickel",
    "Copper",
    "Zinc",
    "Gallium",
    "Germanium",
    "Arsenic",
    "Selenium",
    "Bromine",
    "Krypton",
    "Rubidium",
    "Strontium",
    "Yttrium",
```

```
    "Zirconium",
    "Niobium",
    "Molybdenum",
    "Technetium",
    "Ruthenium",
    "Rhodium",
    "Palladium",
    "Silver",
    "Cadmium",
    "Indium",
    "Tin",
    "Antimony",
    "Tellurium",
    "Iodine",
    "Xenon",
    "Cesium",
    "Barium",
    "Lanthanum",
    "Cerium",
    "Praseodymium",
    "Neodymium",
    "Promethium",
    "Samarium",
    "Europium",
    "Gadolinium",
    "Terbium",
    "Dysprosium",
    "Holmium",
    "Erbium",
    "Thulium",
    "Ytterbium",
    "Lutetium",
    "Hafnium",
    "Tantalum",
    "Tungsten",
    "Rhenium",
    "Osmium",
    "Iridium",
    "Platinum",
    "Gold",
    "Mercury",
    "Thallium",
    "Lead",
    "Bismuth",
    "Polonium",
    "Astatine",
    "Radon",
```

```
    "Francium",
    "Radium",
    "Actinium",
    "Thorium",
    "Protactinium",
    "Uranium",
    "Neptunium",
    "Plutonium",
    "Americium",
    "Curium",
    "Berkelium",
    "Californium",
    "Einsteinium",
    "Fermium",
    "Mendelevium",
    "Nobelium",
    "Lawrencium",
    "Rutherfordium",
    "Dubnium",
    "Seaborgium",
    "Bohrium",
    "Hassium",
    "Meitnerium",
    "Darmstadtium",
    "Roentgenium",
    "Copernicium",
    "Nihonium",
    "Flerovium",
    "Moscovium",
    "Livermorium",
    "Tennessine",
    "Oganesson",
]

def find_longest_name(elements):
  """
  Finds the longest name in a list of chemical elements.

  Args:
    elements: A list of strings representing chemical element names.

  Returns:
    A tuple containing the longest name and its length.
  """
  longest_name = ""
  max_length = 0
```

```
  for element in elements:
    if len(element) > max_length:
      longest_name = element
      max_length = len(element)

  return longest_name, max_length


longest_name, length = find_longest_name(elements_list)

print("The longest element name is: {}".format(longest_name))
print("Its length is: {}".format(length))
```

```
The longest element name is: Rutherfordium
Its length is: 13
```

This program defines a function find_longest_name() that takes a list of element names as input. It iterates through the list, keeping track of the longest name encountered so far. Finally, it returns the longest name and its length.

In the example usage, the list elements_list contains some sample element names. The function is called with this list, and the results are printed using the format() method for clear output.

2.1.5 Q3.

Take your full name as input and make an abbreviation of your name based on the initials of the names. E.g. if the input is "Subhash Chandra Bose", the desired output is "S. C. Bose".

[18]:
```
def abbreviate_name(full_name):
    # Split the name into parts
    name_parts = full_name.split()

    # Abbreviate all parts except the last one
    abbreviated = [f"{part[0].upper()}." for part in name_parts[:-1]]

    # Append the last name without abbreviation
    abbreviated.append(name_parts[-1].capitalize())

    # Join the parts together
    return " ".join(abbreviated)

# Input and Output
full_name = input("Enter your full name: ")
print(f"Your full name is {full_name}")
abbreviated_name = abbreviate_name(full_name)
print("Abbreviated Name:", abbreviated_name)
```

```
Your full name is Cristiano Ronaldo dos Santos Aveiro
Abbreviated Name: C. R. D. S. Aveiro
```

2.1.6 Concept about .split()

The `.split()` method in Python is a powerful tool for string manipulation. Here's how it works:

Purpose:

- **Breaks a string into a list of substrings.**
- **Splits the string based on a specified delimiter.**

Syntax:

```
string.split(separator, maxsplit)
```

- **string:** The string you want to split.
- **separator (optional):**
  - The character or substring used to separate the string.
  - If omitted, whitespace (spaces, tabs, newlines) is used as the separator.
- **maxsplit (optional):**
  - The maximum number of splits to perform.
  - If omitted, all occurrences of the separator are used.

Examples:

1. **Splitting by whitespace:**

```
my_string = "This is an example string"
words = my_string.split()
print(words)  # Output: ['This', 'is', 'an', 'example', 'string']
```

2. **Splitting by a specific character:**

```
sentence = "apple,banana,orange"
fruits = sentence.split(",")
print(fruits)  # Output: ['apple', 'banana', 'orange']
```

3. **Using maxsplit:**

```
line = "field1:value1:field2:value2"
parts = line.split(":", 2)
print(parts)  # Output: ['field1', 'value1', 'field2:value2']
```

Key Points:

- The `.split()` method returns a list of substrings.
- If the separator is not found, the entire string is returned as a single-element list.
- The `maxsplit` argument controls the number of splits, which can be useful for specific parsing needs.

2.1.7 Concept about .upper()

The `.upper()` method in Python is a string method that converts all lowercase characters in a string to uppercase characters and returns the new string. It doesn't modify the original string.

Syntax:

```
string.upper()
```

Parameters:

- The `.upper()` method doesn't take any parameters.

Return Value:

- A new string with all lowercase characters converted to uppercase. If the original string doesn't have any lowercase characters, it returns the original string.

Example:

```
my_string = "hello world"
uppercase_string = my_string.upper()
print(uppercase_string)  # Output: "HELLO WORLD"
```

Key Points:

- The `.upper()` method only affects lowercase alphabetic characters. Other characters like numbers, symbols, and uppercase characters remain unchanged.
- The `.upper()` method is case-insensitive, meaning it doesn't distinguish between different cases when comparing strings. For example, `"Hello"` and `"hello"` are considered the same when compared using `.upper()`.

2.1.8 Concept about .capitalize()

The `.capitalize()` method in Python is a string method that converts the first character of the string to uppercase and the rest of the characters to lowercase.

Key Points:

- **Case Conversion:** It specifically targets the first character for uppercase conversion.
- **Case Normalization:** It can be used to standardize the case of strings, especially when dealing with user input or text that might have inconsistent capitalization.
- **Returns a New String:** The `.capitalize()` method does not modify the original string. It returns a new string with the modified case.

Example:

```
text = "hello world"
capitalized_text = text.capitalize()
print(capitalized_text)  # Output: "Hello world"
```

In this example:

1. The original string is `"hello world"`.
2. The `.capitalize()` method is called on the string.
3. The first character, 'h', is converted to uppercase, and the rest of the characters are converted to lowercase.
4. A new string, `"Hello world"`, is created and stored in the `capitalized_text` variable.

2.1.9 Concept .join() vs .append()

Purpose:

- **.join():**

  - Combines elements of an iterable (like a list or tuple) into a single string.
  - Uses a specified separator to join the elements.
- **.append():**
  - Adds a single element to the end of a list.

Syntax:

- **.join():** `python separator.join(iterable)`
  - `separator`: The string to be used between each element.
  - `iterable`: The list or other iterable containing the elements to be joined.
- **.append():** `python list.append(element)`
  - `list`: The list to which you want to add the element.
  - `element`: The element to be added to the end of the list.

Examples:

- **.join():**

```
my_list = ["apple", "banana", "orange"]
joined_string = ",".join(my_list)
print(joined_string)  # Output: "apple,banana,orange"
```

- **.append():**

```
my_list = ["apple", "banana"]
my_list.append("orange")
print(my_list)  # Output: ["apple", "banana", "orange"]
```

Key Differences:

- **.join()** creates a string, while **.append()** modifies a list in-place.
- **.join()** takes an iterable as input, while **.append()** takes a single element.
- **.join()** uses a separator to combine elements, while **.append()** simply adds the element to the end of the list.

2.1.10 Concept: f-string

f-strings, or formatted string literals, are a powerful feature in Python that simplify string formatting and interpolation. They were introduced in Python 3.6 and offer a concise and readable way to embed expressions and variables directly into strings.

Here's a breakdown of how f-strings work:

Syntax:

- An f-string starts with the letter `f` or `F` followed by a string literal enclosed in single or double quotes.
- Expressions or variables to be embedded are placed within curly braces `{}` inside the string.

Example:

```
name = "Alice"
age = 30
sentence = f"My name is {name} and I am {age} years old."
print(sentence)  # Output: "My name is Alice and I am 30 years old."
```

Key Benefits of f-strings:

- **Readability:** f-strings are often considered more readable than older formatting methods like `%` formatting or the `str.format()` method.
- **Conciseness:** They provide a more concise syntax for string interpolation.
- **Flexibility:** You can embed any valid Python expression within the curly braces, including calculations, function calls, and complex logic.
- **Performance:** f-strings are generally faster than older formatting methods, especially for complex expressions.

Advanced Features:

- **Format Specifiers:** You can use format specifiers within the curly braces to control how the embedded value is formatted. For example:

```
pi = 3.14159
formatted_pi = f"The value of pi is {pi:.2f}"  # Output: "The value of pi is 3.14"
```

- **Expression Evaluation:** f-strings evaluate expressions at runtime, allowing for dynamic string construction.

```
x = 10
y = 5
result = f"The sum of {x} and {y} is {x + y}"  # Output: "The sum of 10 and 5 is 15"
```

2.1.11 Q4

Write a program that prints the first 10 elements of a Fibonacci series 1, 1, 2, 3,5, 8, 13,....., where each element is the sum of the two previous elements (the first two numbers are defined to be 1).

[37]:
```
def fibonacci_series(n):
    if n <= 0:
        return []  # Return an empty list if the input is invalid
    elif n == 1:
        return [1]  # Return [1] if only 1 element is requested
    elif n == 2:
        return [1, 1]  # Return [1, 1] if 2 elements are requested
    else:
        # Initialize the first two numbers of the series
        fib_series = [1, 1]
        # Generate the remaining numbers of the series
        for _ in range(2, n):
            next_number = fib_series[-1] + fib_series[-2]
            fib_series.append(next_number)
        return fib_series

# Define the number of elements to display
num_elements = int(input("Enter the number of Fibonacci elements to display: "))
fib_series = fibonacci_series(num_elements)
print(f"First {num_elements} elements of the Fibonacci series:", fib_series)
```

```
First 10 elements of the Fibonacci series: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

[38]:
```python
# Define the number of elements to display
num_elements = int(input("Enter the number of Fibonacci elements to display: "))
fib_series = fibonacci_series(num_elements)
print(f"First {num_elements} elements of the Fibonacci series:", fib_series)
```

```
First 0 elements of the Fibonacci series: []
```

[39]:
```python
# Define the number of elements to display
num_elements = int(input("Enter the number of Fibonacci elements to display: "))
fib_series = fibonacci_series(num_elements)
print(f"First {num_elements} elements of the Fibonacci series:", fib_series)
```

```
First 1 elements of the Fibonacci series: [1]
```

[40]:
```python
# Define the number of elements to display
num_elements = int(input("Enter the number of Fibonacci elements to display: "))
fib_series = fibonacci_series(num_elements)
print(f"First {num_elements} elements of the Fibonacci series:", fib_series)
```

```
First 2 elements of the Fibonacci series: [1, 1]
```

Q.5 Print the pattern using nested for loops (for n lines):

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

[45]:
```python
# Input for the number of lines
n = int(input("Enter the number of lines: "))

# Generate the pattern
for i in range(1, n + 1):  # Outer loop for each line
    for j in range(1, i + 1):  # Inner loop for numbers on each line
        print(j, end=" ")  # Print numbers on the same line with space
    print()  # Move to the next line
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

- `end=" "` specifies that the print function should not add a newline character after printing the number, keeping the numbers on the same line.
- `print()` This line prints an empty string, effectively moving the cursor to the next line after all numbers for a particular line have been printed.

2.1.12 Q6

Print the pattern using nested for loops (for n lines):

```
1

1 3

1 3 5

1 3 5 7

1 3 5 7 9
```

[49]:
```python
# Number of lines in the pattern
n = 5

# Outer loop for each line
for i in range(1, n + 1):
    # Inner loop to generate and print numbers in each line
    for j in range(1, 2 * i, 2):  # Increment by 2 to get odd numbers
        print(j, end=" ")
    print()  # Move to the next line
```

```
1
1 3
1 3 5
1 3 5 7
1 3 5 7 9
```

In the inner loop, by incrementing the numbers by 2 we are ensuring only odd numbers are printed. The outer loop determines how many numbers are printed in each line.

2.1.13 Q8.

Print the pattern for n lines:

```
1 2 3 4 5

1 2 3 4

1 2 3

1 2

1
```

[50]:
```python
n = 5

for i in range(n, 0, -1):
```

```
    for j in range(i):
        print(j+1, end=" ")
    print()
```

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

2.1.14 Q9

Print the pattern for n lines (mind the left spaces):

```
1 2 3 4 5 4 3 2 1

   2 3 4 5 4 3 2

     3 4 5 4 3

        4 5 4

           5
```

[51]:
```
# Number of lines in the pattern
n = 5

# Outer loop for each line
for i in range(1, n + 1):
    # Print leading spaces
    print(" " * (2 * (i - 1)), end="")

    # Print increasing numbers
    for j in range(i, n + 1):
        print(j, end=" ")

    # Print decreasing numbers
    for j in range(n - 1, i - 1, -1):
        print(j, end=" ")

    # Move to the next line
    print()
```

```
1 2 3 4 5 4 3 2 1
  2 3 4 5 4 3 2
    3 4 5 4 3
      4 5 4
        5
```

This code uses two loops within the outer loop: one for increasing numbers and another for de-

creasing numbers, while the leading spaces are managed by multiplying the current line number by 2.

2.1.15 Python Dictionary: Key Points

1. **Definition**:
   - A dictionary is a mutable, unordered collection in Python.
   - Stores data in key-value pairs.
2. **Syntax**: `my_dict = {"key1": "value1", "key2": "value2"}`
3. **Key Characteristics**:
   - Keys must be **unique** and **immutable** (e.g., strings, numbers, tuples).
   - Values can be of any data type and duplicated.
4. **Accessing Values**: `value = my_dict["key1"]`
5. **Adding/Updating Items**: `my_dict["key3"] = "value3" # Adds or updates a key-value pair`
6. **Removing Items**:
   - `pop()`: Removes an item by key.
   - `popitem()`: Removes the last inserted key-value pair (Python 3.7+).
   - `del`: Deletes a specific key or the entire dictionary.
7. **Methods**:
   - `keys()`: Returns all keys.
   - `values()`: Returns all values.
   - `items()`: Returns all key-value pairs.
   - `get()`: Retrieves a value safely (returns `None` if the key doesn't exist).
8. **Iterating**: `for key, value in my_dict.items(): print(key, value)`
9. **Common Operations**:
   - Check existence: `if "key1" in my_dict`
   - Dictionary length: `len(my_dict)`
10. **Nested Dictionaries**:
    - Dictionaries can contain other dictionaries as values. `nested_dict = {"outer_key": {"inner_key": "value"}}`
11. **Comprehension**:
    - Create a dictionary using comprehension. `square_dict = {x: x**2 for x in range(5)}`
12. **Advantages**:
    - Fast lookups, insertions, and deletions.
    - Flexible data storage.
13. **Use Cases**:

- Mapping relationships (e.g., name to age).
- Storing configurations or JSON-like data.

2.1.16 Q10

Without taking using input create a dictionary with some userid: password pairs as key:value pairs.

i) Now take a userid from the user (use input()). If this userid is in your dictionary, print, ask for password; if the password is also correct print "Welcome to the portal!"; otherwise (if userid or password is wrong) print "Invalid credentials!"

ii) Create a service for password change. Take a userid from the user (use input()). If this userid is in your dictionary, ask for a new password (twice). If the provided passwords are the same (and different from the existing password), update the corresponding password in the dictionary. If the userid is not in your dictionary, print "Invalid username".

[3]:
```
### part 1
# Step 1: Create a dictionary with userid:password pairs
credentials = {
    "user1": "password123",
    "user2": "password456",
    "guest": "guestPass",
    "johnDoe": "john1234"
}

# Step 2: Take userid as input from the user
user_id = input("Enter your user ID: ")

# Step 3: Check if the userid exists in the dictionary
if user_id in credentials:
    # If userid exists, ask for the password
    password = input("Enter your password: ")

    # Step 4: Validate the password
    if credentials[user_id] == password:
        print("Welcome to the portal!")
    else:
        print("Invalid credentials!")
else:
    print("Invalid credentials!")
```

```
Welcome to the portal!
```

[9]:
```
### part 2
# Service for password change
def change_password():
    user_id = input("Enter your user ID for password change: ")

    if user_id in credentials:
```

```
        new_password = input("Enter your new password: ")
        confirm_password = input("Confirm your new password: ")

        if new_password == confirm_password:
            if new_password != credentials[user_id]:
                credentials[user_id] = new_password
                print("Password updated successfully!")
            else:
                print("New password cannot be the same as the old password.")
        else:
            print("Passwords do not match.")
    else:
        print("Invalid username.")

# Example usage of the password change service
change_password()
```

```
Password updated successfully!
```

2.1.17 Q10

Using the Tabulation Method find an interval containing a root of the equation $sin(x) + x^2 - 1$ in the interval [0, 1]. You may use the sin() function in the math module.

[6]:
```
import math

def find_root_interval(f, a, b, step=0.01):
    """
    Finds an interval containing a root of the equation f(x) = 0 within [a, b].

    Parameters:
        f (function): The function f(x) to evaluate.
        a (float): Start of the interval.
        b (float): End of the interval.
        step (float): Step size for tabulation (default: 0.01).

    Returns:
        tuple: A tuple (x1, x2) where f(x1) * f(x2) < 0, indicating a root is␣
 ↪in the interval (x1, x2).
               Returns None if no such interval is found.
    """
    x = a
    while x < b:
        f_x1 = f(x)
        f_x2 = f(x + step)
        if f_x1 * f_x2 < 0:  # Root is in the interval [x, x + step]
            return (x, x + step)
        x += step
```

```
    return None  # No root found in the interval

# Define the function
def f(x):
    return math.sin(x) + x**2 - 1

# Define the interval
a, b = 0, 1

# Call the function to find the interval containing a root
interval = find_root_interval(f, a, b, step=0.01)

if interval:
    print(f"The root is in the interval {interval}")
else:
    print("No root found in the interval [0, 1]")
```

```
The root is in the interval (0.6300000000000003, 0.6400000000000003)
```

2.1.18 Q11

Consider a ball dropping from a height 'h' (value, in cms, taken as input from the user). Each time the ball hits the ground, it bounces up half the immediate previous height it was at. That is after it is dropped from height h, it hits the ground and bounces up height h/2, drops down again, hits the ground, and bounces up height h/4, and so on. Write a program that prints these heights h, h/2, h/4…n times, where n is taken as input from the user. However, if the height reaches 0.05 cm, you should stop immediately. Also, you should print these heights as floating-point numbers using formatted output.

[8]:
```
def bouncing_ball():
    try:
        # Input from the user
        initial_height = float(input("Enter the initial height of the ball in␣
 ↪cm: "))
        n = int(input("Enter the number of bounces to calculate: "))

        if initial_height <= 0 or n <= 0:
            print("Height and number of bounces must be positive numbers.")
            return

        # Initialize variables
        current_height = initial_height

        print("Heights after each bounce:")
        for i in range(n):
            # Print the current height
            print(f"Bounce {i + 1}: {current_height:.2f} cm")
```

```
            # Calculate the next height
            current_height /= 2

            # Stop if the height is less than 0.05 cm
            if current_height < 0.05:
                print("Height has fallen below 0.05 cm. Stopping.")
                break

    except ValueError:
        print("Invalid input. Please enter numeric values for height and␣
 ↪bounces.")

# Call the function
bouncing_ball()
```

```
Heights after each bounce:
Bounce 1: 10.00 cm
Bounce 2: 5.00 cm
Bounce 3: 2.50 cm
Bounce 4: 1.25 cm
Bounce 5: 0.62 cm
Bounce 6: 0.31 cm
Bounce 7: 0.16 cm
Bounce 8: 0.08 cm
Height has fallen below 0.05 cm. Stopping.
```