CN101

Lecture 5 (Part 2)

Lists and Tuples

Topics

- Finding Items in Lists with the `in` Operator
- List Methods and Useful Built-in Functions
- Copying Lists
- Two-Dimensional Lists
- List Comprehension
- Tuples

Finding Items in Lists with the `in` Operator

- You can use the `in` operator to determine whether an item is contained in a list
  - General format: *`item`* `in` *`list`*
  - Returns `True` if the item is in the list, or `False` if it is not in the list
- Similarly you can use the `not in` operator to determine whether an item is not in a list

check_username.py

```
# Define the registered users
registered_users = ["alice", "bob", "charlie", "david"]

# Get the username from the user
username = input("Enter a username to check: ")

# Check if the username is in the registered users
if username in registered_users:
    print(f"Welcome back, {username}!")
else:
    print(f"{username} is not a registered user.")


```

Program Output

```
Enter a username to check: bob
Welcome back, bob!
```

Program Output

```
Enter a username to check: peter
peter is not a registered user.
```

check_shopping_list.py

```
# Define the shopping list
shopping_list = ["milk", "eggs", "bread", "butter"]

# Get the item from the user
item = input("Enter an item to check in the shopping list: ")

# Check if the item is in the shopping list
if item not in shopping_list:
    print(f"{item} is not in your shopping list.")
else:
    print(f"{item} is already in your shopping list.")


```

Program Output

```
Enter an item to check in the shopping list: banana
banana is not in your shopping list.
```

Program Output

```
Enter an item to check in the shopping list: eggs
eggs is already in your shopping list.
```

List Methods

- `append(item)` : used to add items to a list – *item* is appended to the end of the existing list

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers.append(6)
>>> numbers
[1, 2, 3, 4, 5, 6]
```

books_read1.py

```
1  # Example list of books read over a month
2  books_read = []
3
4  # Adding books to the list
5  books_read.append("1984 by George Orwell")
6  books_read.append("To Kill a Mockingbird by Harper Lee")
7  books_read.append("The Great Gatsby by F. Scott Fitzgerald")
8
9  # Display the books read
10 print("Books read this month:")
11 for book in books_read:
12     print(book)
```

Program Output

```
Books read this month:
1984 by George Orwell
To Kill a Mockingbird by Harper Lee
The Great Gatsby by F. Scott Fitzgerald
```

books_read2.py

```
# Example list of books read over a month
books_read = []

# Adding more books based on user input
while True:
    new_book = input("Enter a new book you've read (or 'q' to quit): ")
    if new_book == 'q':
        break
    books_read.append(new_book)

# Display the updated list of books
print("Updated list of books read this month:")
for book in books_read:
    print(book)
```

Program Output

```
Enter a new book you've read (or 'q' to quit): The Martian
Enter a new book you've read (or 'q' to quit): Dune
Enter a new book you've read (or 'q' to quit): q
Updated list of books read this month:
The Martian
Dune
```

List Methods (cont'd.)

- `index(item)`: used to determine where an item is located in a list
  - Returns the index of the first element in the list containing `item`
  - Raises `ValueError` exception if *item* not in the list

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers.index(3)
2
>>> numbers.index(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 6 is not in list
```

student_index.py

```
1  student_list = [
2      "Alice",
3      "Bob",
4      "Charlie",
5      "David",
6      "Eve"
7  ]
8
9  student_to_find = input("Enter the name of the student to find their
   position: ")
10
11 if student_to_find in student_list:
12     index = student_list.index(student_to_find)
13     print(f"The student '{student_to_find}' is at position {index}.")
14 else:
15     print(f"The student '{student_to_find}' is not in the class list.")
```

Program Output

```
Enter the name of the student to find their position: Bob
The student 'Bob' is at position 1.
```

Program Output

```
Enter the name of the student to find their position: Peter
The student 'peter' is not in the class list.
```

List Methods (cont'd.)

- `insert(index, item)`: used to insert *item* at position *index* in the list
- `sort()`: used to sort the elements of the list in ascending order

```
>>> numbers = [1, 3, 2, 6, 4]
>>> numbers.insert(2, 5)
>>> numbers
[1, 3, 5, 2, 6, 4]
>>>
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4, 5, 6]
```

to-do_list.py

```
# Initial list of tasks
tasks = [
    "Buy groceries",
    "Clean the house",
    "Finish the report",
    "Call Alice",
    "Pay bills"
]

print("Initial to-do list:")
for task in tasks:
    print(f' - {task}')

# Insert a new task at a specific position
position = int(input("Enter the position to insert the new task (0-based index): "))
new_task = input("Enter the new task to insert: ")
tasks.insert(position, new_task)

print("\nTo-do list after insertion:")
for task in tasks:
    print(f' - {task}')

```

to-do_list.py (cont'd.)

```
23 # Sort the tasks alphabetically
24 tasks.sort()
25
26 print("\nTo-do list after sorting:")
27 for task in tasks:
28     print(f' - {task}')
```

Program Output

```
Initial to-do list:
  - Buy groceries
  - Clean the house
  - Finish the report
  - Call Alice
  - Pay bills
Enter the position to insert the new task (0-based index): 2
Enter the new task to insert: Schedule meeting

To-do list after insertion:
  - Buy groceries
  - Clean the house
  - Schedule meeting
  - Finish the report
  - Call Alice
  - Pay bills
```

Program Output (cont'd.)

```
To-do list after sorting:
  - Buy groceries
  - Call Alice
  - Clean the house
  - Finish the report
  - Pay bills
  - Schedule meeting
```

List Methods (cont'd.)

- `remove(item)`: removes the first occurrence of *item* in the list
  - Raises `ValueError` exception if *item* not in the list
- `reverse()`: reverses the order of the elements in the list

```
>>> numbers = [1, 2, 3, 2, 5]
>>> numbers.remove(2)
>>> numbers
[1, 3, 2, 5]
>>>
>>> numbers.reverse()
>>> numbers
[5, 2, 3, 1]
```

task_list.py

```python
# Initial list of project tasks
tasks = [
    "Design the UI",
    "Develop the backend",
    "Write documentation",
    "Test the application",
    "Deploy to production"
]

print("Initial list of tasks:")
for i, task in enumerate(tasks, 1):
    print(f" {i}. {task}")

# Prompt the user to input the number of the task to remove
task_number = int(input("Enter the number of the task to remove: "))

# Validate the task number
if 1 <= task_number <= len(tasks):
    task_to_remove = tasks[task_number - 1]
    tasks.remove(task_to_remove)
    print(f"\nTask '{task_to_remove}' has been removed.")
else:
    print(f"\nInvalid task number: {task_number}")

```

task_list.py (cont’d.)


```
25 print("\nList of tasks after removal:")
26 for i, task in enumerate(tasks, 1):
27     print(f" {i}. {task}")
28
29 # Reverse the order of tasks
30 tasks.reverse()
31
32 print("\nList of tasks after reversing:")
33 for i, task in enumerate(tasks, 1):
34     print(f" {i}. {task}")
```

Program Output

```
Initial list of tasks:
  1. Design the UI
  2. Develop the backend
  3. Write documentation
  4. Test the application
  5. Deploy to production
Enter the number of the task to remove: 3

Task 'Write documentation' has been removed.

List of tasks after removal:
  1. Design the UI
  2. Develop the backend
  3. Test the application
  4. Deploy to production

List of tasks after reversing:
  1. Deploy to production
  2. Test the application
  3. Develop the backend
  4. Design the UI
```

The `enumerate` function

- he `enumerate` function in Python is used to iterate over a list (or any iterable) and simultaneously get the index of each item along with the item itself.
- Syntax:

```
enumerate(iterable, start=0)
```

  - `iterable`: The sequence you want to iterate over (e.g., list, tuple, string).
  - `start`: The starting index (optional, default is 0).

The `enumerate` function (cont'd.)

```
>>> fruits = ['apple', 'banana', 'cherry']
>>>
>>> for index, fruit in enumerate(fruits):
...     print(index, fruit)
...
0 apple
1 banana
2 cherry
```

```
>>> fruits = ['apple', 'banana', 'cherry']
>>>
>>> for index, fruit in enumerate(fruits, start=1):
...     print(index, fruit)
...
1 apple
2 banana
3 cherry
```

Useful Built-in Functions

- `del` statement: removes an element from a specific index in a list
  - General format: `del` *`list[i]`*

```
>>> numbers = [1, 2, 3, 4, 5]
>>> del numbers[3]
>>> numbers
[1, 2, 3, 5]
>>> del numbers[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

appointments.py

```python
# Initial list of appointments for a week
appointments = [
    "Monday: Doctor's appointment at 10:00 AM",
    "Tuesday: Team meeting at 2:00 PM",
    "Wednesday: Lunch with Sarah at 12:30 PM",
]

print("Initial list of appointments:")
for i, appointment in enumerate(appointments, 1):
    print(f"{i}. {appointment}")

# Prompt the user to input the number of the appointment to delete
appointment_number = int(input("Enter the number of the appointment to delete: "))

# Validate the appointment number
if 1 <= appointment_number <= len(appointments):
    del appointments[appointment_number - 1]
    print(f"\nAppointment number {appointment_number} has been deleted.")
else:
    print(f"\nInvalid appointment number: {appointment_number}")

print("\nList of appointments after deletion:")
for i, appointment in enumerate(appointments, 1):
print(f"{i}. {appointment}")
```

Program Output

```
Initial list of appointments:
1. Monday: Doctor's appointment at 10:00 AM
2. Tuesday: Team meeting at 2:00 PM
3. Wednesday: Lunch with Sarah at 12:30 PM
Enter the number of the appointment to delete: 2

Appointment number 2 has been deleted.

List of appointments after deletion:
1. Monday: Doctor's appointment at 10:00 AM
2. Wednesday: Lunch with Sarah at 12:30 PM
```

Useful Built-in Functions (cont'd.)

- `min` and `max` functions: built-in functions that returns the item that has the lowest or highest value in a sequence
  - The sequence is passed as an argument
- `Sum` functions: built-in functions that returns the sum of all values in a sequence

```
>>> my_list = [5, 4 ,3, 2, 50, 40, 30]
>>> print(f'The lowest value is {min(my_list)}')
The lowest value is 2
>>> print(f'The highest value is {max(my_list)}')
The highest value is 50
>>> print(f'The sum is {sum(my_list)}')
The sum is 134
```

temperature2.py

```python
# Initial list of daily temperatures over a week
temperatures = [22.5, 24.0, 19.8, 21.3, 25.0]

print("Initial daily temperatures:")
for i, temp in enumerate(temperatures, 1):
    print(f"Day {i}: {temp}°C")

# Adding new temperature readings
while True:
    new_temp = input("Enter a new temperature reading (or 'q' to quit): ")
    if new_temp == 'q':
        break
    temperatures.append(float(new_temp))

print("\nUpdated daily temperatures:")
for i, temp in enumerate(temperatures, 1):
    print(f"Day {i}: {temp}°C")

# Analyzing temperature data
min_temp = min(temperatures)
max_temp = max(temperatures)
avg_temp = sum(temperatures) / len(temperatures)

```

temperature2.py (cont'd.)

```
24 print(f"\nMinimum temperature: {min_temp}°C")
25 print(f"Maximum temperature: {max_temp}°C")
26 print(f"Average temperature: {avg_temp:.2f}°C")
```

Program Output

```
Initial daily temperatures:
Day 1: 22.5°C
Day 2: 24.0°C
Day 3: 19.8°C
Day 4: 21.3°C
Day 5: 25.0°C
Enter a new temperature reading (or 'q' to quit): 21.1
Enter a new temperature reading (or 'q' to quit): 25.5
Enter a new temperature reading (or 'q' to quit): q

Updated daily temperatures:
Day 1: 22.5°C
Day 2: 24.0°C
Day 3: 19.8°C
Day 4: 21.3°C
Day 5: 25.0°C
Day 6: 21.1°C
Day 7: 25.5°C
```

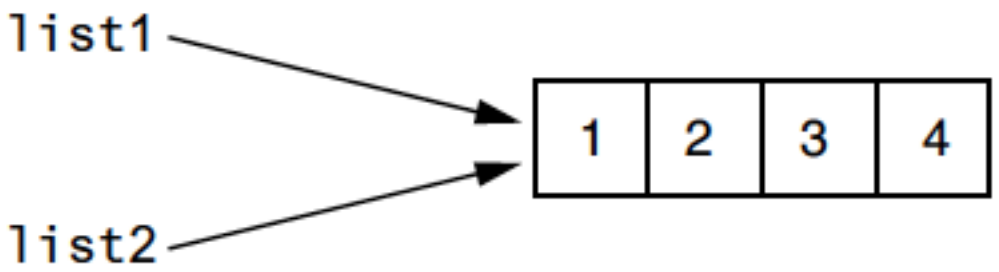
Program Output (cont'd.)

```
Minimum temperature: 19.8°C
Maximum temperature: 25.5°C
Average temperature: 22.60°C
```

List Referencing

```
>>> # Create a list
>>> list1 = [1, 2, 3, 4]
>>>
>>> # Assign the list to the list2 variable
>>> list2 = list1
```

- After this code executes, both variables list1 and list2 will reference the same list in memory.



List Referencing (cont'd.)

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = list1
>>> list1
[1, 2, 3, 4]
>>> list2
[1, 2, 3, 4]
>>>
>>> list1[0] = 99
>>> list1
[99, 2, 3, 4]
>>> list2
[99, 2, 3, 4]
```

inventory.py

```
# Initial inventory list of products in the store
inventory = ["Apple", "Banana", "Orange", "Grape", "Mango"]

# Display items to customers (referencing the same inventory list)
display_items_list = inventory

print("Items available:")
for item in display_items_list:
    print(f"- {item}")
print()

# Simulating a customer buying an item
purchased_item = input("Enter the name of the item the customer wants to buy: ")
if purchased_item in inventory:
    inventory.remove(purchased_item)
    print(f"\nCustomer purchased: {purchased_item}")
else:
    print(f"\n{purchased_item} is not available in the inventory.")

# Display items to customers after the purchase
print("Items available after purchase:")
for item in display_items_list:
    print(f"- {item}")
print()
```

Program Output

```
Items available:
- Apple
- Banana
- Orange
- Grape
- Mango

Enter the name of the item the customer wants to buy: Apple

Customer purchased: Apple
Items available after purchase:
- Banana
- Orange
- Grape
- Mango
```

Copying Lists

- To make a copy of a list you must copy each element of the list
  - Two methods to do this:
    - Creating a new empty list and using a `for` loop to add a copy of each element from the original list to the new list

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = []
>>> for item in list1:
...     list2.append(item)
...
>>> list2
[1, 2, 3, 4]
```

Copying Lists (cont'd.)

- Creating a new empty list and concatenating the old list to the new empty list

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = [] + list1
>>> list2
[1, 2, 3, 4]
```

- As a result, `list1` and `list2` will reference two separate but identical lists.

discounted_prices.py

```
# Initial list of product prices
prices = [10.0, 20.5, 40.9, 50.0, 100.0]

# Display the original prices
print("Original Prices:")
for index, price in enumerate(prices, 1):
    print(f"Product {index}: ${price:,.2f}")
print()

# Copy prices list into two discounted prices lists for comparison
discounted_prices_1 = [] + prices
discounted_prices_2 = [] + prices

# Get the first discount rate from the user
prompt = "Enter first discount rate (as a percentage, e.g., 10 for 10%): "
discount_rate_1 = float(input(prompt)) / 100

# Get the second discount rate from the user
prompt = "Enter second discount rate (as a percentage, e.g., 15 for 15%): "
discount_rate_2 = float(input(prompt)) / 100

# Apply the first discount to each price in the discounted_prices_1 list
for index in range(len(discounted_prices_1)):
    discounted_prices_1[index] = discounted_prices_1[index] * \
                                 (1 - discount_rate_1)
```

discounted_prices.py (cont'd.)

```
26
27 # Apply the second discount to each price in the discounted_prices_2 list
28 for index in range(len(discounted_prices_2)):
29 discounted_prices_2[index] = discounted_prices_2[index] * \
30                               (1 - discount_rate_2)
31
32 # Display the discounted prices for the first discount rate
33 print("Discounted Prices (First Discount Rate):")
34 for index, price in enumerate(discounted_prices_1, 1):
35     print(f"Product {index}: ${price:,.2f}")
36 print()
37
38 # Display the discounted prices for the second discount rate
39 print("Discounted Prices (Second Discount Rate):")
40 for index, price in enumerate(discounted_prices_2, 1):
41     print(f"Product {index}: ${price:,.2f}")
42 print()
43
44 # Display the original prices again to show they are unchanged
45 print("Original Prices After Discount Applied:")
46 for index, price in enumerate(prices, 1):
47     print(f"Product {index}: ${price:,.2f}")
48 print()
```

Program Output

```
Original Prices:
Product 1: $10.00
Product 2: $20.50
Product 3: $40.90
Product 4: $50.00
Product 5: $100.00

Enter first discount rate (as a percentage, e.g., 10 for 10%): 10
Enter second discount rate (as a percentage, e.g., 15 for 15%): 50
Discounted Prices (First Discount Rate):
Product 1: $9.00
Product 2: $18.45
Product 3: $36.81
Product 4: $45.00
Product 5: $90.00

Discounted Prices (Second Discount Rate):
Product 1: $5.00
Product 2: $10.25
Product 3: $20.45
Product 4: $25.00
Product 5: $50.00
```

Program Output (cont'd.)

```
Original Prices After Discount Applied:
Product 1: $10.00
Product 2: $20.50
Product 3: $40.90
Product 4: $50.00
Product 5: $100.00
```

barista_pay.py

```
# NUM_EMPLOYEES is used as a constant for the size of the list.
NUM_EMPLOYEES = 6

# Create a list to hold employee hours.
hours = [0] * NUM_EMPLOYEES

# Get each employee's hours worked.
for index in range(NUM_EMPLOYEES):
    prompt = f'Enter the hours worked by employee {index + 1}: '
    hours[index] = float(input(prompt))

# Get the hourly pay rate.
pay_rate = float(input('Enter the hourly pay rate: '))

# Display each employee's gross pay.
for index in range(NUM_EMPLOYEES):
    gross_pay = hours[index] * pay_rate
    print(f'Gross pay for employee {index + 1}: ${gross_pay:,.2f}')
```

Program Output

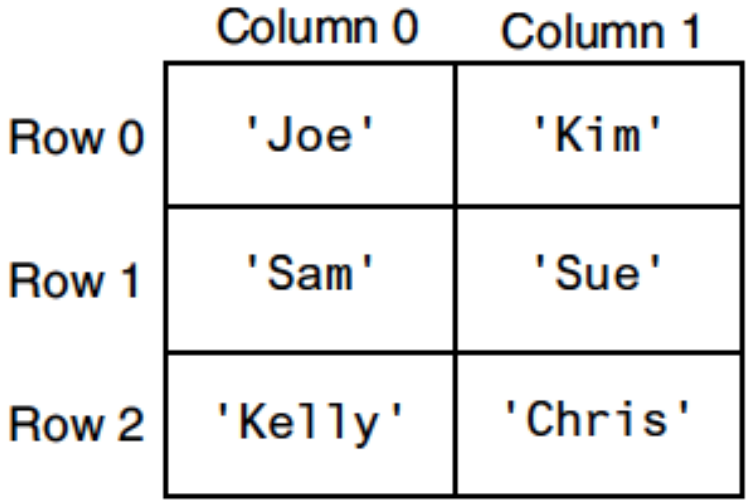
```
Enter the hours worked by employee 1: 10
Enter the hours worked by employee 2: 20
Enter the hours worked by employee 3: 15
Enter the hours worked by employee 4: 40
Enter the hours worked by employee 5: 20
Enter the hours worked by employee 6: 18
Enter the hourly pay rate: 12.75
Gross pay for employee 1: $127.50
Gross pay for employee 2: $255.00
Gross pay for employee 3: $191.25
Gross pay for employee 4: $510.00
Gross pay for employee 5: $255.00
Gross pay for employee 6: $229.50
```

Two-Dimensional Lists

- Two-dimensional list: a list that contains other lists as its elements
  - Also known as nested list
  - Common to think of two-dimensional lists as having rows and columns
  - Useful for working with multiple sets of data
- To process data in a two-dimensional list need to use two indexes
- Typically use nested loops to process

Two-Dimensional Lists (cont'd.)

```
>>> students = [['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']]
>>> students
[['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']]
>>>
>>> students[0]
['Joe', 'Kim']
>>> students[1]
['Sam', 'Sue']
>>> students[2]
['Kelly', 'Chris']
>>>
>>> students[0][0]
'Joe'
```



Two-Dimensional Lists (cont'd.)

```
scores = [[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]]
```

	Column 0	Column 1	Column 2
Row 0	scores[0][0]	scores[0][1]	scores[0][2]
Row 1	scores[1][0]	scores[1][1]	scores[1][2]
Row 2	scores[2][0]	scores[2][1]	scores[2][2]

Two-Dimensional Lists (cont'd.)

```
>>> # Example of a 2D list representing a 3x3 grid
>>> grid = [
...     [1, 2, 3],
...     [4, 5, 6],
...     [7, 8, 9]
... ]
>>>
>>> grid
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
>>> grid[0]
[1, 2, 3]
>>> grid[-1]
[7, 8, 9]
>>> grid[-1][-1]
9
```

fill_2d_list.py

```
# Constants for rows and columns
ROWS = 3
COLS = 4

# Create a two-dimensional list.
values = [[0, 0, 0, 0],
          [0, 0, 0, 0],
          [0, 0, 0, 0]]

# Fill the list with numbers.
i = 1
for r in range(ROWS):
    for c in range(COLS):
        values[r][c] = i
        i += 1

# Display the numbers.
print(values)

```

Program Output

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Example: cn101_scores.py

- The program manages and analyzes scores for different sections of a course (CN101). It collects section names and scores from the user, calculates the average score for each section, prints the scores and averages for each section, and identifies the maximum score across all sections.
- This helps in evaluating the performance of students in various sections and identifying the highest score achieved.

cn101_scores.py pseudocode

```
BEGIN
    INPUT no_sections
    DECLARE section_scores AS list

    FOR each section from 1 to no_sections DO
        INPUT section_name
        DECLARE scores AS list
        PRINT "Input scores for section_name section (type -1 to stop):"
        DECLARE count AS 1

        WHILE True DO
            INPUT score
            IF score == -1 THEN
                BREAK
            END IF
            APPEND score TO scores
            INCREMENT count
        END WHILE

        INSERT section_name AT BEGINNING OF scores
        APPEND scores TO section_scores
    END FOR
```

cn101_scores.py pseudocode

```
DECLARE section_averages AS list

FOR each section_score IN section_scores DO
    SET section_name TO section_score[0]
    SET scores TO section_score[1:]
    CALCULATE average AS sum(scores) / len(scores)
    APPEND (section_name, average) TO section_averages
END FOR

FOR each section_score IN section_scores DO
    SET section_name TO section_score[0]
    SET scores TO section_score[1:]
    PRINT section_name, scores
END FOR

FOR each section_average IN section_averages DO
    SET section_name TO section_average[0]
    SET average TO section_average[1]
    PRINT section_name, "average score:", average
END FOR
```

cn101_scores.py pseudocode

```
    DECLARE all_scores AS list

    FOR each section_score IN section_scores DO
        APPEND section_score[1:] TO all_scores
    END FOR

    SET max_score TO max(all_scores)
    PRINT "Maximum score:", max_score
END
```

cn101_scores.py

```
no_sections = int(input('Input number of CN101 sections: '))

# Input scores for each section.
section_scores = []
for _ in range(no_sections):
    section_name = input('Input CN101 section name: ')
    scores = []
    print(f'Input scores for {section_name} section (type -1 to stop):')
    count = 1
    while True:
        score = float(input(f'Input score #{count}: '))
        if score == -1:
            break
        scores.append(score)
        count += 1
    scores.insert(0, section_name)
    section_scores.append(scores)

# Calculates the average score for each section.
section_averages = []
    for section_score in section_scores:
    section_name = section_score[0]
    scores = section_score[1:]
    average = sum(scores) / len(scores)
    section_averages.append((section_name, average))
```

cn101_scores.py (cont'd.)

```
26
27 # Prints the scores for each section.
28 for section_score in section_scores:
29     section_name = section_score[0]
30     scores = section_score[1:]
31     print(f'{section_name}: {scores}')
32
33 # Prints the average score for each section.
34 for section_average in section_averages:
35     section_name = section_average[0]
36     average = section_average[1]
37     print(f'{section_name} average score: {average:.2f}')
38
39 # Finds the maximum score across all sections.
40 all_scores = []
41 for section_score in section_scores:
42     all_scores += section_score[1:]
43 max_score = max(all_scores)
44 print(f'Maximum score: {max_score}')
```

Program Output

```
Input number of CN101 sections: 2
Input CN101 section name: 810001
Input scores for 810001 section (type -1 to stop):
Input score #1: 89.5
Input score #2: 78.4
Input score #3: 69.0
Input score #4: 65.5
Input score #5: 59.8
Input score #6: -1
Input CN101 section name: 740002
Input scores for 740002 section (type -1 to stop):
Input score #1: 88.8
Input score #2: 67.5
Input score #3: 65.5
Input score #4: 55.5
Input score #5: 71.2
Input score #6: 60.5
Input score #7: -1
810001: [89.5, 78.4, 69.0, 65.5, 59.8]
740002: [88.8, 67.5, 65.5, 55.5, 71.2, 60.5]
810001 average score: 72.44
740002 average score: 68.17
Maximum score: 89.5
```

List comprehension

- List comprehensions provide a concise way to create lists in Python. They are more readable and often faster than using traditional for-loop
- Syntax:

```
list1 = [expression for item in iterable]
```

  - **expression**: The expression that gets evaluated and added to the list.
  - **item**: The variable that takes the value of the element from the iterable.
  - **iterable**: A collection of elements (e.g., list, tuple, string) to iterate over.

List comprehension (cont'd.)

- List comprehensions:

```
list1 = [expression for item in iterable]
```

- For loop:

```
list1 = []
for item in iterable:
    list1.append(expression)
```

List comprehension (cont'd.)

```
>>> list1 = []
>>> for num in range(1, 6):
...     list1.append(num)
...
>>> list1
[1, 2, 3, 4, 5]
>>>
>>> list2 = [num for num in range(1, 6)]
>>> list2
[1, 2, 3, 4, 5]
```

List comprehension (cont'd.)

```
>>> list1 = []
>>> for num in range(1, 6):
...     list1.append(num*2)
...
>>> list1
[2, 4, 6, 8, 10]
>>>
>>> list2 = [num*2 for num in range(1, 6)]
>>> list2
[2, 4, 6, 8, 10]
```

cn101_scores2.py

```
no_sections = int(input('Input number of CN101 sections: '))

# Input scores for each section.
section_scores = []
for _ in range(no_sections):
    section_name = input('Input CN101 section name: ')
    scores = []
    print(f'Input scores for {section_name} section (type -1 to stop):')
    count = 1
    while True:
        score = float(input(f'Input score #{count}: '))
        if score == -1:
            break
        scores.append(score)
        count += 1
    scores.insert(0, section_name)
    section_scores.append(scores)

# Calculates and returns the average score for each section.
section_averages = [
    (section_score[0], sum(section_score[1:]) / len(section_score[1:]))
    for section_score in section_scores
]

```

cn101_scores2.py (cont'd.)

```
25 # Prints the scores for each section.
26 for section_score in section_scores:
27     section_name = section_score[0]
28     scores = section_score[1:]
29     print(f'{section_name}: {scores}')
30
31 # Prints the average score for each section.
32 for section_name, average in section_averages:
33     print(f'{section_name} average score: {average:.2f}')
34
35 # Finds and returns the maximum score across all sections.
36 max_score = max([
37     score
38     for section_score in section_scores
39     for score in section_score[1:]
40 ])
41
42 print(f'Maximum score: {max_score}')
```

Nested List Comprehensions

```
>>> list1 = []
>>> for x in [1, 2]:
...     for y in [3, 4]:
...         list1.append([x, y])
...
>>> list1
[[1, 3], [1, 4], [2, 3], [2, 4]]
>>>
>>> list2 = [[x, y] for x in [1, 2] for y in [3, 4]]
>>> list2
[[1, 3], [1, 4], [2, 3], [2, 4]]
```

Using if Condition in List Comprehensions

- List comprehensions can include an optional if condition to filter items from the iterable before applying the expression
- Syntax:

```
list1 = [expression for item in iterable if condition]
```

  - **`expression`**: The expression that gets evaluated and added to the list.
  - **`item`**: The variable that takes the value of the element from the iterable.
  - **`iterable`**: A collection of elements (e.g., list, tuple, string) to iterate over.
  - **`condition`**: A filtering condition that evaluates to True or False. Only items that meet this condition are processed by the expression.

Using if Condition in List Comprehensions (cont'd.)

```
>>> evens = [num for num in range(10) if num % 2 == 0]
>>> evens
[0, 2, 4, 6, 8]
>>>
>>> greater_5 = [x for x in range(10) if x > 5]
>>> greater_5
[6, 7, 8, 9]
>>>
>>> intersec = [num for num in evens if num in greater_5]
>>> intersec
[6, 8]
```

employee_filter.py

```python
# Data structure containing departments and their employees
departments = [
    [("Alice", 28), ("Bob", 34)], # HR Department
    [("Charlie", 32), ("David", 25)], # IT Department
    [("Eve", 29), ("Frank", 38)] # Finance Department
]

# Nested list comprehension to get employee names above 30
employee_names_above_30 = [
    name
    for department in departments
    for name, age in department
    if age > 30
]

# Print the result
print("Employees above 30 years old:", employee_names_above_30)

```

Program output

```
Employees above 30 years old: ['Bob', 'Charlie', 'Frank']
```

Tuples

- Tuple: an immutable sequence
  - Very similar to a list
  - Once it is created it cannot be changed
  - Format: `tuple_name = (item1, item2)`
  - Tuples support operations as lists
    - Subscript indexing for retrieving elements
    - Methods such as `index`
    - Built in functions such as `len, min, max, sum`
    - Slicing expressions
    - The `in`, `+`, and `*` operators

Tuples (cont'd.)

- Tuples do not support the methods:
  - `append`
  - `remove`
  - `insert`
  - `reverse`
  - `sort`
- Tuples do not support **`del`** statement

```
>>> my_tuple = (1, 2, 3, 4, 5)
>>> my_tuple
(1, 2, 3, 4, 5)
>>>
>>> names = ('Holly', 'Warren', 'Ashley')
>>> for name in names:
...     print(name)
...
Holly
Warren
Ashley
>>>
>>> names = ('Somsak', 'Somsri', 'Somchai')
>>> for i in range(len(names)):
...     print(names[i])
...
Somsak
Somsri
Somchai
```

Tuples (cont'd.)

- Advantages for using tuples over lists:
  - Processing tuples is faster than processing lists
  - Tuples are safe
  - Some operations in Python require use of tuples
- `list()` function: converts tuple to list
- `tuple()` function: converts list to tuple

Note

- If you want to create a tuple with just one element, you must write a trailing comma after the element's value, as shown here:

```
my_tuple = (1, )  # Creates a tuple with one element.
```

- If you omit the comma, you wil not create a tuple. For example, the following statement simply assigns the integer value 1 to the `value` varaiable:

```
Value = (1) # Creates an integer.
```

student_averages.py

```python
# The data structure is a list of tuples, where each tuple contains
# a student's name, math score, and science score
students = [
    ("Alice", 85, 78),
    ("Bob", 70, 82),
    ("Charlie", 90, 95),
    ("David", 65, 70),
    ("Eve", 88, 92)
]

# Calculate the average score of each student.
student_averages = [
    (name, (math + science) / 2)
    for name, math, science in students
]

# Create a list of students who have an average score above 75.
above_75_students = [
    name
    for name, average in student_averages
    if average > 75
]

```

student_averages.py (cont'd.)

```
# Find the student with the highest average score.
highest_average_student = student_averages[0]
for student in student_averages:
    if student[1] > highest_average_student[1]:
        highest_average_student = student

# Print the results
print("Average scores of each student:")
for name, average in student_averages:
    print(f"{name}: {average:.2f}")

print("\nStudents with an average score above 75:")
print(above_75_students)

print(f"\nStudent with the highest average score: " +
      f"{highest_average_student[0]} with an average of " +
      f"{highest_average_student[1]:.2f}")
```

Program Output

```
Average scores of each student:
Alice: 81.50
Bob: 76.00
Charlie: 92.50
David: 67.50
Eve: 90.00

Students with an average score above 75:
['Alice', 'Bob', 'Charlie', 'Eve']
```

employee_contact_search.py

```python
# List of employees where each employee is represented as a tuple.
# (Employee ID, Employee Name, Email Address, Phone Number)
employees = [
    (1, "Alice", "alice@example.com", "123-456-7890"),
    (2, "Bob", "bob@example.com", "987-654-3210"),
    (3, "Charlie", "charlie@example.com", "555-123-4567"),
    (4, "David", "david@example.com", "444-555-6666"),
    (5, "Eve", "eve@example.com", "111-222-3333")
]

# Define the partial name to search for
search_name = input("Enter the partial name of the employee: ")
# Define what to search for (phone or email)
search_type = input("Do you want to search for 'phone' or 'email'? ")

if search_type == "phone":
    results = [
        (employee[1], employee[3])
        for employee in employees
        if search_name in employee[1]
    ]
    result_type = "Phone number"
```

employee_contact_search.py (cont'd.)

```
elif search_type == "email":
    results = [
        (employee[1], employee[2])
        for employee in employees
            if search_name in employee[1].lower()
    ]
    result_type = "Email address"
else:
    print("Invalid search type. Please choose 'phone' or 'email'.")
    results = []
    result_type = ""

# Print the results
if results:
    for name, contact in results:
        print(f"{result_type} of {name}: {contact}")
else:
    print(f"No employee found with the partial name '{search_name}'")
```

Program Output

```
Enter the partial name of the employee: Bob
Do you want to search for 'phone' or 'email'? email
Email address of Bob: bob@example.com
```

Program Output

```
Enter the partial name of the employee: ch
Do you want to search for 'phone' or 'email'? phone
Phone number of Charlie: 555-123-4567
```

Program Output

```
Enter the partial name of the employee: a
Do you want to search for 'phone' or 'email'? email
Email address of Alice: alice@example.com
Email address of Charlie: charlie@example.com
Email address of David: david@example.com
```

Program Output

```
Enter the partial name of the employee: b
Do you want to search for 'phone' or 'email'? emal
Invalid search type. Please choose 'phone' or 'email'.
```

Summary

- This chapter covered:
  - Lists, including:
    - Repetition and concatenation operators
    - Indexing
    - Techniques for processing lists
    - Slicing and copying lists
    - List methods and built-in functions for lists
    - Two-dimensional lists
    - List Comprehension
  - Tuples, including:
    - Immutability
    - Difference from and advantages over lists