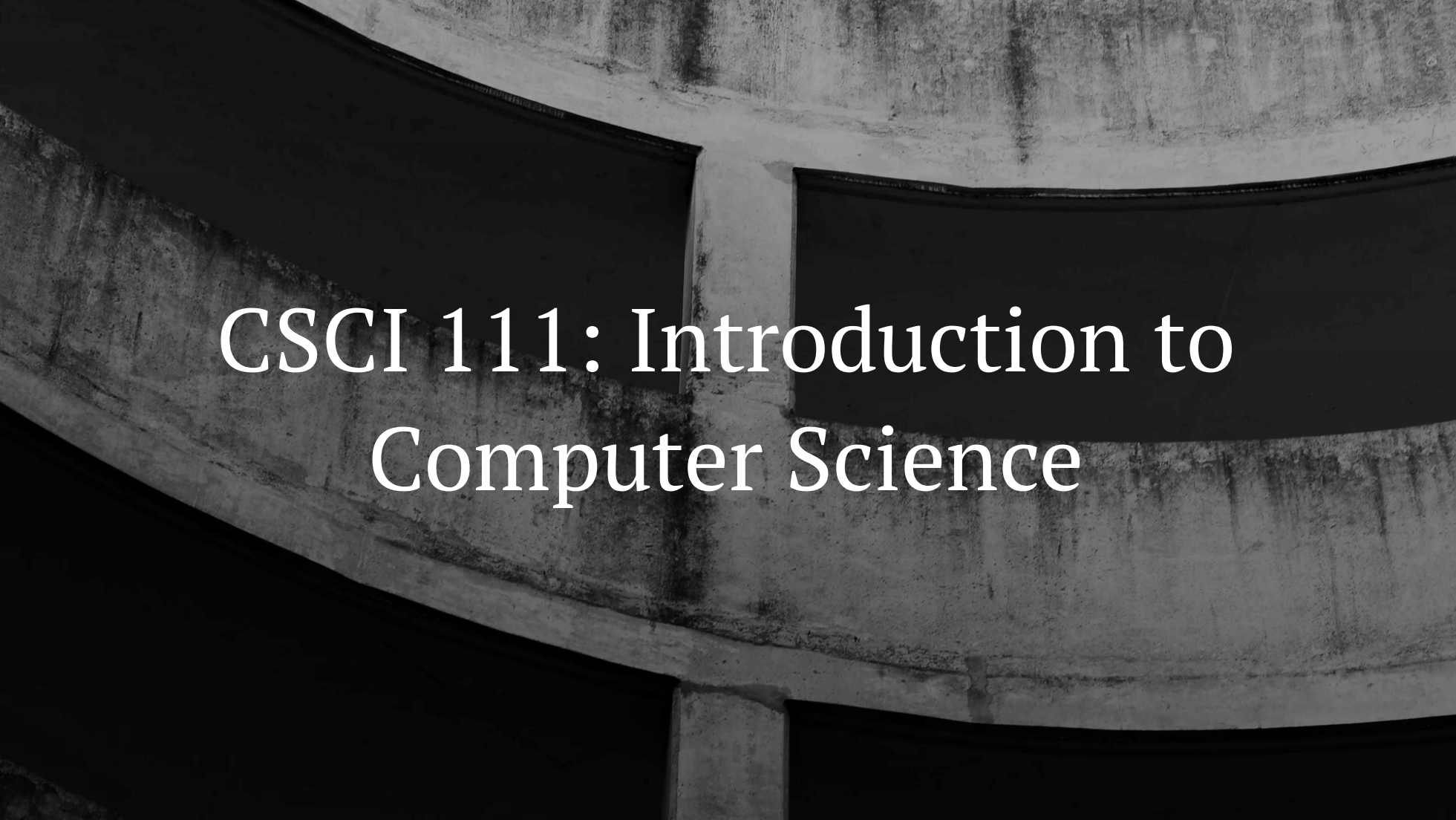

CSCI 111: Introduction to Computer Science

Introduction to Python Collections

- **Overview**: Collections manage groups of objects in Python. Types include Lists, Sets, and Tuples.
- **Use Cases**: Lists for ordered items, sets for unique items, tuples for immutable sequences.

Lists in Python

- **Mutability**: Mutable (elements can be added, removed, or changed).
- **Ordering**: Ordered (elements can be indexed by a range of numbers).
- **Syntax**: Created with square brackets, e.g., `[1, 2, 3]`.
- **Use Cases**: Ideal for scenarios where the order of elements matters and modifications are frequent, such as managing a dynamic list of items in an inventory or a todo list.

Sets in Python

- **Mutability**: Mutable (elements can be added or removed, but not directly changed).
- **Ordering**: Unordered (elements do not have a fixed position; their order is not guaranteed).
- **Syntax**: Created with curly braces, e.g., `{1, 2, 3}` or `set()` for an empty set.
- **Use Cases**: Best used when you need to maintain a collection of unique items and order is irrelevant, such as storing a collection of distinct identifiers or properties.

Tuples in Python

- **Mutability**: Immutable (once created, elements cannot be changed).
- **Ordering**: Ordered (elements can be indexed by a range of numbers).
- **Syntax**: Created with parentheses, e.g., `(1, 2, 3)` or a trailing comma for a single element `(1,)`.
- **Use Cases**: Suitable for storing records that should not change, such as database entries or fixed configurations, and for ensuring data integrity.

Common Features of Python Collections

- **Iterable**: All types are iterable, making them usable in loops and comprehensions.
- **Element Type Diversity**: Can contain elements of various types, including mixed types within the same collection.
- **Built-in Functionality**: Support a variety of built-in methods and functions, which can be explored using `help()` and `dir()`.

Using `help()` and `dir()`

- **Tools for Discovery**: Use `help(list)` and `dir(set)` to explore methods and properties.
- **Example**: `help(tuple)` shows tuple methods like count and index.

Creating Lists

- **Basic Creation**: `my_list = [1, 2, 3, 4]`.
- **Empty List**: `empty_list = []`.
- **Mixed Types**: `mixed_list = [1, 'two', 3.0, [4, 5]]`.

Accessing List Elements

- **Indexing**: Access an item by its position - `first_item = my_list[0]`.
- **Negative Indexing**: Access from the end - `last_item = my_list[-1]`.

Adding Elements to Lists

- **Using** `append()` : Add an item to the end - `my_list.append(5)` .
- **Using** `insert()` : Insert an item at a specified index - `my_list.insert(1, 'inserted')` .
- **Using** `extend()` : Merge another list or iterable to the end - `my_list.extend([6, 7, 8])` .

Removing Elements from Lists

- **Using** `remove()` : Remove the first occurrence of an item - `my_list.remove('inserted')` .
- **Using** `pop()` : Remove an item at a given index and return it - `popped_item = my_list.pop(0)` .
- **Clearing the List**: Remove all items - `my_list.clear()` .

Basic List Methods

- **Finding Elements**: `index = my_list.index('two')` finds the index of 'two'.
- **Counting Elements**: `count = my_list.count(2)` counts how many times 2 appears.
- **Reversing the List**: `my_list.reverse()` reverses the elements in place.

Sorting Lists

- **Sorting in Place**: `my_list.sort()` sorts the list in ascending order by default.
- **Custom Sorting**: `my_list.sort(reverse=True)` for descending order.
- **Using `sorted()`**: `sorted_list = sorted(my_list)` creates a new sorted list without altering the original.

Copying Lists

- **Shallow Copy**: `new_list = my_list.copy()` or `new_list = my_list[:]`.
- **Deep Copy**: Required when the list contains lists or other mutable objects, using `import copy` and `deep_copy = copy.deepcopy(my_list)`.

Understanding List Slicing

- **Definition**: Slicing is a method for extracting a subset of elements from a list.
- **Syntax**: `list[start:stop:step]`
  - `start` : Index where the slice starts (inclusive).
  - `stop` : Index where the slice ends (exclusive).
  - `step` : The increment between each index in the slice (default is 1).

Basic Slicing Examples

- **Extract Sublists**: Define a start and stop.
  - **Example**: `numbers = [0, 1, 2, 3, 4, 5]`
  - **Slice**: `sub_numbers = numbers[1:4]`
  - **Result**: `[1, 2, 3]` – Starts at index 1 and stops before index 4.

Negative Indices in Slicing

- **Access Slices from the End**: Use negative indices to count backwards from the end of the list.
  - **Example**: `numbers = [0, 1, 2, 3, 4, 5]`
  - **Negative Slice**: `end_numbers = numbers[-3:-1]`
  - **Explanation**: `-3` refers to the third last item (`3`), and `-1` is one before the last item (`5`), so the slice includes `[3, 4]`.

Slicing with Steps

- **Skip Items in the Slice**: Define a step to create slices with skipped elements.
  - **Example**: `numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - **Stepped Slice**: `every_second_number = numbers[::2]`
  - **Result**: `[0, 2, 4, 6, 8]` – Takes every second element across the entire list.

Reversing Lists with Slicing

- **Reverse a List Quickly**: Use a negative step without start or stop.
  - **Example**: `numbers = [0, 1, 2, 3, 4, 5]`
  - **Reversed**: `reversed_numbers = numbers[::-1]`
  - **Result**: `[5, 4, 3, 2, 1, 0]` – Steps backwards through the whole list.

Omitting Indices in Slices

- **Full Slices**: Omit both start and stop to copy the list.
  - **Example**: `numbers = [0, 1, 2, 3, 4, 5]`
  - **Copy**: `all_numbers = numbers[:]`
  - **Result**: `[0, 1, 2, 3, 4, 5]` – A complete copy of the list, useful for operations that need a full list without altering the original.

Practical Applications of Slicing: Analyzing Data

- **Contextual Understanding**: Slicing makes it easy to segment lists for specific analyses, which is invaluable in fields like meteorology, economics, and accounting.

Temperature Analysis for Lexington, VA

- **Annual Temperatures**: `[31, 34, 42, 53, 62, 71, 75, 74, 67, 55, 45, 35]` (Average high temperatures from January to December).
- **Extracting Summer Temperatures**:
  - **Slice**: `summer_temps = temperatures[5:8]`
  - **Result**: `[71, 75, 74]` – Represents average high temperatures for June, July, and August.

Economic Analysis: Quarterly Earnings

- **Quarterly Revenue**: `[20000, 22000, 21000, 24000, 25000, 26000, 27000, 28000, 29000, 30000, 31000, 32000]` (Monthly revenue over a year for a company).
- **Extracting Q3 Revenue**:
  - **Slice**: `Q3_revenue = quarterly_revenue[6:9]`
  - **Result**: `[27000, 28000, 29000]` – Revenue figures for July, August, and September.

In-Class Assignment: Financial Trend Analysis

Objective

- Analyze quarterly financial data using Python lists to uncover revenue trends and derive insights.

In-Class Assignment: Financial Trend Analysis

Background

- Provided data: Monthly revenue figures for a company over two years.
- Data format: `revenues = [20, 22, 21, 24, 25, 26, 27, 28, 29, 30, 31, 32, 34, 36, 35, 38, 39, 40, 41, 42, 43, 44, 45, 46]` (in thousands of dollars).

Tasks for Financial Trend Analysis

1. Split the Data into Quarters

- **Description**: Transform the monthly revenue data into quarterly data.
- **Task**: Create a list of lists called `quarters`, where each sublist contains three months of revenue data.
- **Hint**: Use a loop to iterate over the `revenues` list and slice every three months. Append each sublist to the `quarters` list.

Tasks for Financial Trend Analysis

2. Calculate Quarterly Growth

- **Description**: Analyze the growth in revenue from one quarter to the next.
- **Task**: Calculate the percentage growth for each quarter, skipping the first quarter as it has no previous quarter for comparison.
- **Formula**: To calculate the growth percentage for quarter `n`, use the formula:
  - Growth% = ((Current Quarter Revenue - Previous Quarter Revenue) / Previous Quarter Revenue) * 100
- **Output**: Store the growth percentages in a new list called `growth_percentages`.

Tasks for Financial Trend Analysis

3. Identify Best and Worst Quarters

- **Description**: Determine which quarters had the highest and lowest growth in revenue.
- **Task**: Find the maximum and minimum values in the `growth_percentages` list.
- **Analysis**: Discuss possible reasons for the highest and lowest growth quarters based on factors like market conditions, seasonal variations, or operational changes.

Introduction to Python Sets

- **Definition**: Sets are collections of unique elements. They are mutable and unordered.
- **Syntax**: Created with curly braces or the `set()` function.
  - Example: `my_set = {1, 2, 3}` or `my_set = set([1, 2, 3])`.
- **Characteristics**: Sets automatically remove duplicates and are useful for operations involving uniqueness.

Creating and Using Sets

- **Creating Sets**: `my_set = {1, 2, 3, 2}` will automatically reduce to `{1, 2, 3}` due to the uniqueness constraint.
- **Empty Set**: `empty_set = set()` because `{}` creates an empty dictionary, not a set.

Adding and Removing Elements

- **Adding Elements**: `my_set.add(4)` adds a new element to the set.
- **Removing Elements**: `my_set.remove(2)` removes an element from the set. If the element is not present, it raises a KeyError.
- **Safe Remove**: `my_set.discard(5)` removes an element if present, but does nothing if the element is not found.

Set Operations

- **Union**: Combines elements from two sets, no duplicates.
  - Syntax: `set1.union(set2)` or `set1 | set2`.
  - Example: `{1, 2} | {2, 3}` results in `{1, 2, 3}`.
- **Intersection**: Elements common to both sets.
  - Syntax: `set1.intersection(set2)` or `set1 & set2`.
  - Example: `{1, 2} & {2, 3}` results in `{2}`.
- **Difference**: Elements in one set but not the other.
  - Syntax: `set1.difference(set2)` or `set1 - set2`.
  - Example: `{1, 2} - {2, 3}` results in `{1}`.
- **Symmetric Difference**: Elements in either set but not in both.
  - Syntax: `set1.symmetric_difference(set2)` or `set1 ^ set2`.
  - Example: `{1, 2} ^ {2, 3}` results in `{1, 3}`.

Checking Set Membership

- **In Operation**: Check if an item is in a set.
  - Example: `2 in {1, 2, 3}` returns `True`.
- **Not In Operation**: Check if an item is not in a set.
  - Example: `4 not in {1, 2, 3}` returns `True`.

Practical Applications of Python Sets

Why Use Sets?

- **Uniqueness**: Sets automatically ensure that all elements are unique, removing any duplicates.
- **Efficiency**: Sets offer fast membership testing, making them ideal for large datasets where checking if an item exists is frequent.

Common Uses of Sets

Removing Duplicates

- **Description**: Sets are excellent for cleaning data to remove duplicate entries quickly.
  - **Example**:

```
data = [1, 2, 2, 3, 4, 4, 4, 5]
unique_data = list(set(data))  # Converts list to set and back to list
print(unique_data)  # Output: [1, 2, 3, 4, 5]
```

Membership Tests

- **Description**: Sets allow for very fast membership testing compared to lists or tuples.
  - **Example**:

```
inventory = set(['apple', 'banana', 'orange'])
print('apple' in inventory)  # True, very fast even with large sets
```

Set Operations for Data Analysis

- **Description**: Use set operations like union, intersection, and difference to analyze datasets.
  - **Union**: Combine items from two data sources, removing duplicates.
  - **Intersection**: Find common items between two datasets.
  - **Difference**: Discover items present in one dataset but not the other.
  - **Example**:

```
students_2019 = {'Alice', 'Bob', 'Charlie'}
students_2020 = {'Bob', 'Charlie', 'Dave'}
alumni = students_2019.intersection(students_2020)
new_students = students_2020.difference(students_2019)
print('Alumni:', alumni)  # Output: {'Bob', 'Charlie'}
print('New Students:', new_students)  # Output: {'Dave'}
```

Advanced Set Usage

Mathematical Problems

- **Description**: Sets can solve problems involving subsets, supersets, and disjoint sets.
  - **Example**:

```
A = {1, 2, 3}
B = {3, 4, 5}
C = {1, 2, 3, 4, 5}
print(A.issubset(C))  # True
print(B.isdisjoint(A))  # False
```

Data Deduplication in Real-time Systems

- **Description**: Use sets to manage unique user sessions or connections in real-time applications such as chat servers or online games.
  - **Example**:

```
active_users = set()
def login(user_id):
    active_users.add(user_id)
def logout(user_id):
    active_users.discard(user_id)
```

Summary and Best Practices

- **Efficiency**: Use sets when you need to ensure uniqueness or perform frequent membership tests.
- **Limitations**: Sets are unordered and only store hashable items, so they cannot contain mutable objects like lists.
- **Practicality**: Sets are invaluable for data analysis, real-time systems, and any application where the dataset's integrity is key.

Introduction to Python Tuples

- **Definition**: Tuples are immutable collections of items.
- **Syntax**: Created with parentheses or no brackets at all.
  - Example: `my_tuple = (1, 2, 3)` or simply `my_tuple = 1, 2, 3`.
- **Characteristics**: Once a tuple is created, it cannot be altered, which makes tuples hashable and suitable for use as keys in dictionaries.

Creating and Using Tuples

- **Creating Tuples**: `point = (1, 2)` or `point = 1, 2` for a coordinate pair.
- **Empty Tuple**: `empty_tuple = ()`.
- **Single Element Tuple**: `singleton = (1,)` — note the comma is necessary.

Accessing Tuple Elements

- **Indexing**: `point = (1, 2, 3); print(point[0])` will output `1`.
- **Slicing**: `point = (1, 2, 3, 4); print(point[1:3])` will output `(2, 3)`.

Tuple Immutability

- **Immutable Nature**: Once created, elements cannot be changed.
  - Trying to alter an element: `point[1] = 3` will raise a TypeError.
- **Benefits**: Immutability makes tuples predictable and safe from modification.

Tuple Packing and Unpacking

- **Packing**: `tuple = 1, 2, 3` — elements are packed into a tuple.
- **Unpacking**: `x, y, z = tuple` — elements are extracted into variables `x`, `y`, and `z`.
- **Extended Unpacking**: `x, *y = tuple` — first element to `x`, remaining to list `y`.

Tuple Methods

- **Limited Methods**: Due to immutability, tuples have fewer methods.
  - `count()` : `tuple.count(1)` — returns the number of times `1` appears in the tuple.
  - `index()` : `tuple.index(1)` — returns the first index of the value `1` .

Practical Applications of Tuples

Why Use Tuples?

- **Data Integrity**: Tuples are perfect for situations where the integrity and immutability of the data are critical. Since tuples are immutable, they ensure that data cannot be changed inadvertently, which is crucial in settings where consistency and data safety are priorities.

Common Uses of Tuples

- **Function Arguments and Returns**: Tuples are commonly used to pass a fixed group of related values from or to a function. For example, returning multiple values from a function neatly without needing a class or structure.
  - **Example**:

```
def min_max(items):
    return min(items), max(items)  # returns a tuple of minimum and maximum
```

- **Fixed Data Records**: Tuples can store data records where modification is not required, such as a set of coordinates or the attributes of a database entry that should not change after creation.
  - **Example**:

```
coordinates = (13.4125, 103.8667)  # Latitude and Longitude of Angkor Wat
```

- **Looping Through Combined Data**: You can use tuples for looping through pairs or groups of items. This is particularly useful in data analysis where each element of the tuple represents a different dimension or metric.
  - **Example**:

```
for name, age in [('Alice', 32), ('Bob', 48), ('Charlie', 28)]:
    print(f"{name} is {age} years old.")
```

Advantages Over Lists

- **Performance**: Tuples can be slightly faster than lists when iterating through large amounts of data.
- **Safety**: Using tuples prevents accidental data modification, which can be critical in multi-threaded environments where data consistency must be maintained.