Lecture slides - Week 5

Programming in Python - Data Structures

Dr. Aamir Akbar

Director of both AWKUM AI Lab and AWKUM Robotics, Final Year Projects (FYPs) coordinator, and lecturer at the department of Computer Science
Abdul Wali Khan University, Mardan (AWKUM)

Contents

Introduction

Introduction to Data Structures

- **Data Types**: Data types define the nature of data stored in a variable. They specify the range of values that a variable can hold and the operations that can be performed on it. For example, in Python, you have data types like integers, floating-point numbers, and booleans, which define how numeric and boolean data is represented and operated upon.
- **Data Structures**: A data structure is a fundamental concept in computer science and programming. Data structures define how data is organized and stored in memory. They determine how collections of data are structured, accessed and manipulated. Data structures can be composed of one or more data types. For instance, a list in Python is a data structure that can hold elements of various data types (integers, strings, etc.).

Primitive Data Types in Python

In Python, primitive data types typically refer to the basic data types that come built-in with the language. These include:

- **int**: Represents integers (whole numbers) such as 1, -5, 1000.
- **float**: Represents floating-point numbers (real numbers) with decimal points, like 3.14, -0.5, 2.0.
- **bool**: Represents boolean values, which can be either True or False.
- **str**: Represents strings, which are sequences of characters enclosed in single or double quotes.

Python Data Structures

Strings i

In python, Strings are:

- sequences of characters.
- enclosed in single (' ') or double (" ") quotes.
- immutable (cannot be changed once created).

Example

```
my_string = "Hello, World!"
```

Common String Methods:

1. `len(my_string)` - Returns the length of the string.
2. `my_string.upper()` - Converts the string to uppercase.
3. `my_string.lower()` - Converts the string to lowercase.
4. `my_string.strip()` - Removes leading and trailing whitespace.

Strings ii

5. `my_string.split(delimiter)` - Splits the string into a list of substrings using the specified delimiter.
6. `my_string.find(substring)` - Returns the index of the first occurrence of the substring (or -1 if not found).
7. `my_string.replace(old, new)` - Replaces all occurrences of the old substring with the new substring.

Lists i

In python, Lists

- are ordered collections of elements.
- may have elements that can be of different types.
- are mutable (can be modified).

Example

```
my_list = [1, 2, 3, "apple", "banana"]
```

Common List Methods:

1. `len(my_list)` - Returns the number of elements in the list.
2. `my_list.append(element)` - Adds an element to the end of the list.
3. `my_list.insert(index, element)` - Inserts an element at the specified index.

Lists ii

4. `my_list.remove(element)` - Removes the first occurrence of the specified element.
5. `my_list.pop(index)` - Removes and returns the element at the specified index (or the last element if index is not provided).
6. `my_list.index(element)` - Returns the index of the first occurrence of the specified element (or raises an error if not found).
7. `my_list.count(element)` - Returns the number of occurrences of the specified element in the list.
8. `my_list.sort()` - Sorts the list in ascending order.
9. `my_list.reverse()` - Reverses the order of elements in the list.

Tuples

In Python, Tuples

- are ordered collections like lists.
- are immutable (cannot be changed once created).
- are typically used for grouping related data.

Example

```
m_tuple = (1, 2, 3, "apple", "banana")
```

Common Tuple Methods:

1. `len(my_tuple)` - Returns the number of elements in the tuple.
2. `my_tuple.index(element)` - Returns the index of the first occurrence of the specified element.
3. `my_tuple.count(element)` - Returns the number of occurrences of the specified element in the tuple.

Sets i

In Python, Sets are

- unordered collections of unique elements.
- useful for eliminating duplicate values.

Example

```
my_set = {1, 2, 3, 4, 5}
```

Common Set Methods:

1. `len(my_set)` - Returns the number of elements in the set.
2. `element in my_set` - Checks if an element is present in the set (returns a boolean).
3. `my_set.add(element)` - Adds an element to the set.
4. `my_set.remove(element)` - Removes the specified element from the set (raises an error if not found).
5. `my_set.discard(element)` - Removes the specified element from the set (no error if not found).

Dictionaries i

In Python, Dictionaries

- are collections of key-value pairs.
- keys are unique and used to access values.

Example

```
phone_book = {"name": "Ali", "mobile": "0300-123456",
"age": 30, "city": "Mardan"}
```

Common Dictionary Methods:

1. `len(phone_book)` - Returns the number of key-value pairs in the dictionary.
2. `key in phone_book` - Checks if a key is present in the dictionary (returns a boolean).
3. `phone_book[key]` - Accesses the value associated with a key.

Dictionaries ii

4. `phone_book.keys()` - Returns a list of all keys in the dictionary.
5. `phone_book.values()` - Returns a list of all values in the dictionary.
6. `phone_book.items()` - Returns a list of key-value pairs as tuples.
7. `phone_book.update(other_dict)` - Updates the dictionary with key-value pairs from another dictionary.
8. `phone_book.pop(key)` - Removes the key-value pair with the specified key.

Data Structure Slicing and Comprehension

Slicing i

Slicing in Python allows you to extract a portion of a sequence (e.g., lists, strings, tuples) by specifying a range of indices. The syntax for slicing is as follows:

- `sequence[start:stop:step]`
  - `start`: The starting index (included).
  - `stop`: The ending index (excluded).
  - `step` (optional): The step or interval between elements (default is 1).

Examples:

- Slicing a list:

```
my_list = [1, 2, 3, 4, 5]
sliced_list = my_list[1:4]  # Result: [2, 3, 4]
```

Slicing ii

- Slicing a string:

```
my_string = "Hello, World!"
sliced_string = my_string[0:5]  # Result: "Hello"
```

- Slicing a tuple:

```
my_tuple = (10, 20, 30, 40, 50)
sliced_tuple = my_tuple[1:4]  # Result: (20, 30, 40)
```

- Using step in slicing:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
sliced_list_step = my_list[1:8:2]  # Result: [2, 4, 6, 8]
```

Slicing is a feature that works with ordered sequences like lists, strings, and tuples. Sets are unordered collections of unique elements, and they do not support indexing or slicing because they lack a specific order.

Comprehensions i

Comprehensions in Python are concise and expressive ways to create new sequences (lists, tuples, sets, and dictionaries) by applying an expression to each item in an existing iterable and optionally filtering them. They help make your code more readable and efficient.

- **List Comprehension**:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [x**2 for x in numbers]
# Result: squared_numbers = [1, 4, 9, 16, 25]
```

- **Tuple Comprehension**:

```
numbers = (1, 2, 3, 4, 5)
squared_numbers = tuple(x**2 for x in numbers)
# Result: squared_numbers = (1, 4, 9, 16, 25)
```

Comprehensions ii

- **Set Comprehension**:

```
numbers = {1, 2, 2, 3, 3, 4, 5}
unique_squares = {x**2 for x in numbers}
# Result: unique_squares = {1, 4, 9, 16, 25}
```

- **Dictionary Comprehension**:

```
names = [‘Alice’, ‘Bob’, ‘Charlie’]
name_lengths = {name: len(name) for name in names}
# Result: name_lengths = {‘Alice’: 5, ‘Bob’: 3,
‘Charlie’: 7}
```

Problem Solving Approach with Examples

Problem-Solving Approach: Chat-bot

Problem Statement: Create a Fun Chatbot

Objective: Create a simple chatbot program in Python that can engage in conversations with users and respond to specific prompts with fun and engaging messages.

Requirements: The chatbot should be able to respond to the following prompts:

1. hello
2. how are you
3. tell me a joke
4. tell me a fact
5. what is your favorite color
6. bye

The chatbot should handle prompts in a case-insensitive manner, e.g, it should respond to "Hello," "HELLO," and "hello" in the same way.