Data Structures and Algorithms

@DSA2425

Introduction to Data Structures and Python Basics

Course Outline and Key Concepts

Haitham A. El-Ghareeb

Mansoura University / Faculty of Computers and Information Sciences

September 29, 2024

Course Overview

- Understanding Data Structures and Algorithms.
- Introduction to Python programming.
- Building efficient and scalable software solutions.
- Hands-on exercises for practical learning.

Why Study Data Structures and Algorithms?

- Core to efficient problem-solving and programming.
- Crucial for managing and organizing data.
- Key to writing optimized and performant code.
- Widely used in technical interviews and industry scenarios.

Key Concepts

Data Structure: Organized way to store and manipulate data (e.g., lists, stacks, queues).

Algorithm: A sequence of steps to solve a problem (e.g., sorting, searching).

Programming Language: A tool to implement data structures and algorithms (e.g., Python, C++).

Python Control Flow: Conditionals

If-Else Statement Example:

```
# Check if a course is active
course_name = "Data Structures"
is_active = True

if is_active:
    print(f"The course {course_name} is active.")
else:
    print(f"The course {course_name} is not active.")
```

Explanation: Checks if the course is active and prints a message accordingly.

Python Control Flow: For Loop

For Loop Example:

```
# List of course modules
modules = ["Introduction", "Linked Lists",
           "Stacks", "Queues", "Trees"]

for module in modules:
    print(f"Module: {module}")
```

Explanation: Iterates over the list of modules and prints each one.

Python Control Flow: While Loop

While Loop Example:

```
# Count the number of enrolled students
enrolled_students = 0

while enrolled_students < 10:
    enrolled_students += 1
    print(f"Enrolled Student Count: {enrolled_students}")
```

Explanation: Keeps a count of enrolled students until it reaches 10.

Object-Oriented Programming (OOP)

- OOP models real-world entities as classes and objects.
- Allows for better organization and modular code.
- Supports principles like encapsulation, inheritance, and polymorphism.

Classes and Objects in Python

Example: Course Class

```python
class Course:
    def __init__(self, name, course_id, credits):
        self.name = name
        self.course_id = course_id
        self.credits = credits

    def display_info(self):
        print(f"Course: {self.name}, ID: {self.course_id}")

# Create a Course object
course = Course("Data Structures", 101, 4)
course.display_info()
```

Explanation: Defines a course with its properties and a method to display its information.

Encapsulation in Python

Example: Private Attributes

```python
class Student:
    def __init__(self, name, student_id):
        self.__name = name  # Private attribute
        self.__student_id = student_id

    def display_info(self):
        print(f"Student: {self.__name}, ID: {self.__student_id}")

# Create a Student object
student = Student("Alice", 1001)
student.display_info()
```

Explanation: Uses double underscores to make attributes private.

Inheritance in Python

Example: Person and Instructor

```python
# Base class
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, I am {self.name}")

# Subclass
class Instructor(Person):
    def __init__(self, name, instructor_id):
        super().__init__(name)
        self.instructor_id = instructor_id

    def display_info(self):
        print(f"Instructor: {self.name}, ID: {self.instructor_id}")
```

Polymorphism in Python

Example: Different Login Methods

```
# Base class
class User:
    def login(self):
        print("User logged in")

# Subclasses
class Student(User):
    def login(self):
        print("Student logged in")

class Instructor(User):
    def login(self):
        print("Instructor logged in")

# Polymorphic behavior
users = [Student(), Instructor()]
```

Understanding Time Complexity

- Measures the time an algorithm takes as a function of input size.
- Represented using **Big O notation**.
- Examples: $O(1)$ - constant time, $O(n)$ - linear time, $O(n^2)$ - quadratic time.

Examples of Time Complexity

O(1): Constant Time

```
def is_even(number):
    return number % 2 == 0
```

O(n): Linear Time

```
def sum_elements(numbers):
    total = 0
    for num in numbers:
        total += num
    return total
```

Understanding Space Complexity

- Measures the memory an algorithm uses relative to input size.
- Examples: $O(1)$ - constant space, $O(n)$ - linear space.
- Important for optimizing memory usage in large-scale applications.

Interactive Learning Resources

- **Videos**: Explore practical Python examples and tutorials.
- **Egyptian Knowledge Bank (EKB)**: Access e-books, articles, and course-related materials.
- Develop self-study habits to enhance your programming knowledge.

Accessing EKB Resources

1. Register at `https://www.ekb.eg/`.
2. Activate your account using the email confirmation link.
3. Access course-related content at `https://mffeci.ekb.eg/linkresolver/openurl/v0.1`.

- Explore additional e-books, articles, and journals to support your learning.

Encouraging Self-Learning

- Programming evolves rapidly – stay current with new technologies.
- Cultivate a habit of self-learning and explore beyond the syllabus.
- Utilize multiple resources to deepen your understanding of concepts.

Conclusion

- We covered the importance of data structures and algorithms.
- Introduced Python basics and Object-Oriented Programming.
- Discussed time and space complexities using Big O notation.

Let's make this learning journey an engaging and rewarding one!