Table of Contents

Instructional Objectives (reStart Program)

Day 1

Python and Data Structures

Understand Basic Data Structures

- Define fundamental data structures such as lists, tuples, sets, and dictionaries.
- Explain the characteristics and use cases of each data structure.
- Demonstrate how to create, access, modify, and iterate through these data structures.

Explore Advanced Data Structures

- Introduce more complex data structures such as stacks, queues, heaps, and trees.
- Discuss the characteristics, operations, and applications of each advanced data structure.
- Illustrate how to implement these data structures using Python.

Understand Time and Space Complexity

- Define time complexity and space complexity.
- Analyze the time and space complexity of common operations on various data structures.
- Discuss the importance of choosing appropriate data structures based on their performance characteristics.

Apply Data Structures in Problem Solving

- Solve programming problems using appropriate data structures.
- Practice implementing algorithms and data structures to solve real-world problems.
- Analyze the efficiency and effectiveness of different data structure choices in problem-solving scenarios.

Optimize Data Structure Usage

- Optimize the usage of data structures to improve performance and memory usage.
- Identify opportunities for using built-in Python data structures and libraries.
- Apply best practices for choosing and implementing data structures in Python applications.

Suggested Tutorials

[Optimizing Python Code with Data Structures] https://learnscripting.org/optimizing-performance-strategies-for-data-structure-efficiency-in-python/
[Implementing Data Structures in Python] https://realpython.com/tutorials/data-structures/
[Data Structures and Algorithms in Python] https://www.geeksforgeeks.org/data-structures
[Big O Notation and Time Complexity in Python]
https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

Questions and Answers

How would you choose between a list and a set for storing unique elements?
Discuss the advantages of using a default dict over a standard dictionary in Python.
Implement a caching mechanism using a dictionary to optimize repetitive computations.
How do you implement a stack using a list in Python?
Explain the FIFO (First-In-First-Out) principle in queues.
What is a binary tree, and how is it different from a binary search tree?
How would you use a dictionary to count the frequency of elements in a list?
Solve the problem of finding the maximum subarray sum using dynamic programming.
Implement a priority queue and use it to solve a scheduling problem.
What is the time complexity of searching for an element in a list?
Explain the space complexity of a recursive function.
Compare the time complexity of sorting algorithms such as bubble sort and merge sort.
What are the differences between lists and tuples in Python?
How do you add an element to a set in Python?
Explain how to access a value in a dictionary by its key.
Create a Caesar encryption/description program

Day 2

Object Oriented Programming

Understand OOP Concepts

- Define the principles of OOP, including encapsulation, inheritance, and polymorphism.
- Explain the advantages of using OOP in software development.
- Demonstrate how classes, objects, attributes, and methods are used in Python.

Create and Use Classes

- Define and implement classes to model real-world entities and concepts.
- Describe class attributes, instance attributes, and methods.
- Illustrate how to instantiate objects from classes and access their attributes and methods.

Explore Inheritance and Polymorphism

- Implement inheritance to create hierarchies of related classes.
- Demonstrate method overriding and method overloading for polymorphic behavior.
- Discuss the benefits of code reusability and extensibility through inheritance.

Implement Encapsulation and Abstraction

- Encapsulate data and behavior within classes to protect and control access.
- Use access modifiers such as public, private, and protected to enforce encapsulation.
- Apply abstraction to hide implementation details and expose only essential interfaces.

Practice Design Patterns and SOLID Principles

- Apply design patterns such as Singleton, Factory, and Observer to solve common design problems.
- Follow SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) for better code design.
- Discuss the importance of design patterns and SOLID principles in creating maintainable, scalable, and testable code.

Suggested Tutorials

[Object-Oriented Programming (OOP) in Python] https://realpython.com/python3-object-oriented-programming
[Python Classes and Objects] https://www.geeksforgeeks.org/python-classes-and-objects
[Inheritance and Polymorphism in Python] https://realpython.com/inheritance-composition-python

[Encapsulation and Abstraction in Python] https://stackabuse.com/encapsulation-and-abstraction-in-python
[Supercharge your Classes with Super] https://realpython.com/python-super/
[Design Patterns in Python] https://refactoring.guru/design-patterns/python

Questions and Answers

- What is encapsulation, and how does it improve code organization and maintainability?
- Explain the difference between a class and an object in Python.
- Give an example of polymorphism using method overriding in Python.
- Define a class `Rectangle` with attributes `length` and `width`, and a method `calculate_area()` to compute the area.
- Instantiate an object of the `Rectangle` class with a length of 5 and a width of 3, and calculate its area.
- Explain the difference between a class attribute and an instance attribute.
- Define a base class `Animal` with a method `make_sound()` and subclasses `Dog` and `Cat` that override this method.
- Illustrate how to use polymorphism to call the `make_sound()` method on different objects.
- Discuss the concept of method overloading and how it differs from method overriding.
- Define a class `BankAccount` with private attributes `balance` and `account_number`, and methods to deposit, withdraw, and get balance.
- Demonstrate how encapsulation prevents direct access to private attributes from outside the class.
- Explain how abstraction can be achieved by providing high-level methods for common operations.
- Implement the Singleton pattern to create a logger class with a single instance shared across the application.
- Discuss how the Open/Closed principle can be applied to extendable software design.
- Provide an example of violating the Liskov Substitution principle and suggest a refactor to adhere to it.

File I/O and Database Management

Understanding File Objects

- Learn how to open files in Python using the built-in `open()` function.
- Understand different file modes such as read mode ('r'), write mode ('w'), append mode ('a'), and read/write mode ('r+').
- Explore the concept of file objects and how they are used to interact with files.

Reading from Files
- Learn how to read the contents of a file using methods like `read()`, `readline()`, and `readlines()`.

- Understand how to handle large files efficiently by reading them line by line.

Writing to Files

- Learn how to write data to files using the `write()` method.
- Understand how to handle different data types such as strings and numbers when writing to files.

File Management

- Explore file management operations such as creating, renaming, and deleting files and directories using the `os` module.
- Learn how to check for the existence of files and directories using functions like `os.path.exists()`.

Using Context Managers

- Understand the concept of context managers and how they can be used with files using the `with` statement.
- Learn how context managers automatically handle opening and closing files, ensuring proper resource management.

Database Integration

- Learn how to integrate file I/O with databases such as PostgreSQL using libraries like Psycopg2 or SQLAlchemy.
- Understand how to read data from and write data to a PostgreSQL database using Python.

Suggested Tutorials

[Real Python - Reading and Writing Files in Python] https://realpython.com/read-write-files-python
[Python Docs - File I/O] https://docs.python.org/3/tutorial/inputoutput.htmlreading-and-writing-files

Questions and Answers

- How do you open a file named "data.txt" in read mode?
- What is the difference between the `read()` and `readline()` methods?
- How do you write the string "Hello, World!" to a file named "output.txt"?
- Explain the purpose of the `with` statement when working with files.
- How can you check if a file exists before attempting to open it for reading or writing?
- How do you establish a connection to a PostgreSQL database in Python?

Day 3

Modules, Regex and Typing

Python Modules

- Understand the concept of modules in Python and how they facilitate code organization and reusability.
- Learn how to create and import custom modules in Python.
- Explore commonly used built-in modules in Python and their functionalities.

Regular Expressions (Regex)

- Understand the basics of regular expressions and their role in pattern matching and text processing.
- Learn how to use regex to search, match, and manipulate strings in Python.
- Explore advanced regex techniques such as grouping, capturing, and lookahead/lookbehind assertions.

Typing in Python

- Understand the purpose of type annotations and how they enhance code readability and maintainability.
- Learn how to use type hints to specify variable types, function parameters, and return values in Python code.
- Explore type checking tools such as `mypy` and `pyright` and understand how they help catch type-related errors in Python code.

Suggested Tutorials

[Python Modules] https://realpython.com/python-modules-packages
[Creating and Using Python Modules]
https://www.learnpython.org/en/Modules_and_Packages
[Python Standard Library] https://docs.python.org/3/library/index.html
[Regular Expression HOWTO] https://docs.python.org/3/howto/regex.html
[RegexOne - Learn Regular Expressions with Simple, Interactive Exercises]
https://regexone.com
[Python Regex Cheat Sheet] https://www.dataquest.io/blog/regex-cheatsheet
[Typing in Python] https://realpython.com/python-type-checking
[Python Typing Docs] https://docs.python.org/3/library/typing.html
[Type Checking Python with MyPy] https://realpython.com/python-type-checking/

Questions and Answers

- Explain the difference between built-in modules and third-party modules in Python
- Demonstrate how to import a function from a custom module and use it in another Python script.
- What are some examples of built-in modules in Python, and what purposes do they serve?
- Describe the syntax and purpose of the `re.match()` function in Python's `re` module.
- Write a regular expression pattern to match email addresses in a given string.
- Explain the difference between greedy and non-greedy matching in regular expressions, and provide an example.
- How do type annotations benefit developers during code development and maintenance?
- Write a Python function that calculates the area of a rectangle and add type annotations to it.

Day 4

Python Naming Conventions, PEP8, Exporting Modules/Submodules/Classes, and Data Transfer Objects (DTO)

Understand Python Naming Conventions

Explain the standard naming conventions for variables, functions, classes, constants, and modules in Python.
Write Python code that follows the standard naming conventions, enhancing code readability and maintainability.

Learn PEP8 Standards

Understand and apply the PEP8 guidelines for Python code style.
Format Python code according to PEP8, ensuring consistency and readability across different codebases.

Exporting and Importing Modules and Submodules

Demonstrate how to create, export, and import modules and submodules in Python.
Structure Python projects using modules and submodules, promoting code organization and reuse.

Defining and Using Classes

Explain how to define classes and use them in Python.
Students will be able to create and utilize classes to implement object-oriented programming principles in their projects.

Understand and Implement Data Transfer Objects (DTO)

Explain the concept of Data Transfer Objects and how they are used to transfer data between different parts of an application.
Create and use DTOs to encapsulate data and improve the structure of applications.

Suggested Tutorials

Tutorial 1: Understanding Python Naming Conventions

1. Naming Conventions:

  - Variables: `lowercase_with_underscores`
  - Functions: `lowercase_with_underscores`
  - Classes: `CamelCase`
  - Constants: `UPPERCASE_WITH_UNDERSCORES`
  - Modules and Packages: `lowercase_with_underscores`

Exercise:
- Write a Python script that includes examples of variables, functions, classes, and constants, following the naming conventions.

Tutorial 2: Learning PEP8 Standards

1. PEP8 Guidelines:
   - Indentation: 4 spaces per indentation level.
   - Line Length: Limit lines to 79 characters.
   - Blank Lines: Use blank lines to separate top-level functions and class definitions.
   - Imports: Imports should usually be on separate lines.

2. Using linters:
   - Tools like `flake8` or `pylint` to check for PEP8 compliance.

Exercise:
- Write a Python script and use a linter to ensure it adheres to PEP8 guidelines.

Tutorial 3: Exporting and Importing Modules and Submodules

1. Creating Modules:
   - Example of a simple module:

```
 mymodule.py
def my_function():
    print("Hello from mymodule")
```

2. Creating Submodules:
   - Example of a package with submodules:

```
mypackage/
    __init__.py
    submodule1.py
    submodule2.py
```

3. Importing Modules and Submodules:

Importing a module
```
import mymodule
mymodule.my_function()
```

Importing a submodule
```
from mypackage import submodule1
submodule1.some_function()
```

Exercise:
- Create a package with at least two submodules and demonstrate importing and using them in a script.

Tutorial 4: Defining and Using Classes

1. Defining a Class:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def display_value(self):
        print(self.value)
```

2. Using a Class:

```
obj = MyClass(10)
  obj.display_value()
```

Exercise:
- Define a class with at least one method and one attribute. Create an instance of the class and call its method.

Tutorial 5: Understanding and Implementing DTOs

1. What is a DTO?:
  - A DTO is an object that carries data between processes. It reduces the number of method calls and simplifies the data transfer.

2. Creating a DTO:

```
class UserDTO:
    def __init__(self, username, email):
        self.username = username
        self.email = email
```

3. Using a DTO:

```
user_dto = UserDTO(username="john_doe", email="john@example.com")
```

Exercise:
- Create a DTO class for a simple user profile and demonstrate using it to transfer data between functions.

Questions and Answers

What naming convention should you use for class names in Python?

CamelCase (e.g., `MyClass`).
What tool can you use to check for PEP8 compliance in your code?
`flake8` or `pylint`.
How do you import a function `foo` from a submodule `submod` within a package `mypack`?
`from mypack.submod import foo`
What is the purpose of a Data Transfer Object (DTO)?
To encapsulate data for transfer between processes, reducing method calls and simplifying data management.
Write a simple class definition in Python.

```
class SampleClass:
    def __init__(self, value):
        self.value = value

    def show_value(self):
        print(self.value)
```

Additional Resources

[PEP8 - Style Guide for Python Code] https://www.python.org/dev/peps/pep-0008/
[PyLint Documentation] https://pylint.pycqa.org/en/latest/
[Flake8 Documentation] https://flake8.pycqa.org/en/latest/
[Python Packaging Authority] https://pypi.org/
[Python OOP Tutorial] https://realpython.com/python3-object-oriented-programming/

Day 5

Developing with FastAPI, Uvicorn, and Swagger

Understand FastAPI Basics

Explain the fundamental concepts of FastAPI and its benefits for web development.
Create simple FastAPI applications and understand its basic features.

Learn to Use Uvicorn with FastAPI

Demonstrate how to use Uvicorn as an ASGI server to run FastAPI applications.
Set up and run FastAPI applications using Uvicorn.

Explore Swagger Integration in FastAPI

Explain how FastAPI integrates with Swagger to provide interactive API documentation.
Use Swagger UI to test and document their FastAPI applications.

Develop and Document APIs with FastAPI

Guide students through creating and documenting APIs using FastAPI and Swagger.
Develop and document RESTful APIs with FastAPI and explore them using Swagger.

Suggested Tutorials

Tutorial 1: Understanding FastAPI Basics

1. Introduction to FastAPI:
  - Benefits: High performance, easy to use, modern features (async support, automatic docs, etc.)
  - Installation:

```
pipenv install fastapi
```

2. Creating a Simple FastAPI Application:

```
from fastapi import FastAPI

  app = FastAPI()

  @app.get("/")
  def read_root():
      return {"Hello": "World"}

  @app.get("/items/{item_id}")
  def read_item(item_id: int, q: str = None):
      return {"item_id": item_id, "q": q}
```

Exercise:
- Write a simple FastAPI application with two endpoints: one that returns a greeting message and another that returns an item based on an ID.

Tutorial 2: Using Uvicorn with FastAPI

1. Introduction to Uvicorn:
  - Uvicorn is a lightning-fast ASGI server for Python 3.7+.

2. Running FastAPI with Uvicorn:
  - Installation:

```
pipenv install uvicorn
```

  - Running the application:

```
uvicorn myapp:app --reload
```

Exercise:
- Install Uvicorn and run your FastAPI application using Uvicorn. Test the endpoints to ensure they are working.

Tutorial 3: Exploring Swagger Integration in FastAPI

1. Automatic Documentation:
  - FastAPI provides automatic interactive API documentation using Swagger UI.

2. Accessing Swagger UI:
  - By default, available at http://127.0.0.1:8000/docs

3. Exploring API with Swagger:
  - Use the interactive documentation to test API endpoints.

Exercise:
- Open the Swagger UI for your FastAPI application and test the endpoints. Observe how FastAPI automatically generates documentation based on your code.

Tutorial 4: Developing and Documenting APIs with FastAPI

1. Creating and Documenting Endpoints:
  - Define more complex endpoints with parameters and responses.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
```

```
        price: float
        tax: float = None

    @app.post("/items/", response_model=Item)
    def create_item(item: Item):
        return item

    @app.get("/items/{item_id}", response_model=Item)
    def read_item(item_id: int):
        return {"item_id": item_id, "name": "Item Name", "price": 100.0}
```

2. Error Handling and Validation:
   - Using FastAPI's validation and error handling features.

Exercise:
- Extend your FastAPI application to include a POST endpoint that accepts an `Item` and returns it. Ensure that Swagger UI reflects these changes.

Questions and Answers

What command is used to install FastAPI?
pipenv install fastapi`

How do you run a FastAPI application using Uvicorn?
uvicorn main:app --reload

What URL is used to access Swagger UI for a FastAPI application running locally?
http://127.0.0.1:8000/docs`

How does FastAPI automatically generate API documentation?
FastAPI uses Pydantic models and type annotations to generate interactive API documentation with Swagger UI.

Write an example of a FastAPI endpoint that uses a Pydantic model for request validation.

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

@app.post("/items/")
def create_item(item: Item):
    return item
```

Additional Resources

[FastAPI] https://fastapi.tiangolo.com/

[Uvicorn] https://www.uvicorn.org/
[Swagger UI] https://swagger.io/tools/swagger-ui/
[Pydantic] https://pydantic-docs.helpmanual.io/
[Real Python FastAPI] https://realpython.com/fastapi-python-web-apis/