Object-oriented and Functional Programming in Python Language, Libraries and modern Design Patterns

Course originator: Dr. Daniel J. Duffy, Datasim Education BV

Summary of the Modules
The course consists of six modules. Each module deals with a major well-defined topic.

Module 1: Essential Python for those with no experience of programming or of Python.
Module 2: Software design principles and their realisation in Python.
Module 3: Object-Oriented Programming (OOP) in Python.
Module 4: Functional Programming (FP) in Python.
Module 5: Advanced Python Topics.
Module 6: System Development and Design Patterns in Python applications.

This course can be taken on two levels: first, for beginners and novices, we advise modules 1 to 4 while modules 5 and 6 are for more experienced programmers as well as for those who feel confident with the first four modules.
The course starts officially March 15 2022. It is possible to register at any time.

Module 1 Preamble, Learning to break fall and Foundations
The goal of this module is to introduce a number of fundamental programming features and syntax and how they are supported in Python. In particular, we discuss vital concepts such as variables, functions and program control structure and we show how they are implemented in Python by taking simple but at the same time illustrative examples. These examples will be extended and generalised in later modules. In this sense we take an incremental approach by building up expertise using already developed results and code.

It is assumed that you have installed Python on your system and that you are comfortable with the Python tools and IDE that you are using. All you need is basically an environment in which you can create Python code and then compile and run it. The focus is on learning to become a good Python programmer and preparing the student for later models.

By completing this module you will have gained essential and fundamental hands-on Python experience. The analogy is learning how to break fall before taking judo lessons.

We realise that many students do not necessarily have a formal education in Computer Science (CS) and for this reason we add a number of sections on data structure, algorithms and other important concepts that help in your understanding of Python.

Variables and Functions
- Lexical Structure
- Python Strings
- Fundamental Data Types
- Functions
- Code layout rules

Program Structure
- Scope and Lifetime of Variables
- Control Flow Structure
- Looping and Iteration
- Iterables, Iterators and Tuples

Fundamental data types
- Numbers and Numeric Operations
- Testing for valid numbers
- Analysis of floating-point numbers

Control flow statements
- If and if-else statements
- While statement
- For statement
- Iterators

Expressions and Operators
- What is an expression?
- Operators and operator precedence in expressions
- Numeric operations and conversions
- Sequence operations

Data Types
- Vectors and lists
- Deque, queue and priority queue
- Sets
- Associative arrays and dictionaries

Algorithm Analysis
- Asymptotic behaviour of functions
- Asymptotic order; orders of complexity
- Hierarchy of orders
- Order relationships
- Hard Problems

Types of Algorithms
- Searching and sorting
- Merge
- Insertion, remove
- Linear and binary search

Module 2 Background and Design Foundations

The goal of this very important module (all modules are important) is to introduce and discuss the three major design philosophies/paradigms/styles that are supported in Python. It is important to understand them and to use them to build robust and flexible applications:
- Object oriented style: programs built around classes, inheritance and composition.
- Functional style: programs built based on stateless functions.
- Modular programming: decompose a problem into loosely coupled modules.

We introduce these concepts by discussing them in such a way that provides insights into which style or combination of styles are most suitable for a given set of requirements. To this end, we integrate these concepts into the slides, sample code and exercises as we progress in the course. In this case we take a range of examples, starting with simple cases before moving to more advanced and closer to real-life examples as elaborated in later modules.

Module 2 prepares the way for Modules 3 and 4. In particular, module 1 discusses the object-oriented and functional programming paradigms from a language-independent, correct and rigorous perspective. An extra advantage is that the topics in this module are based on the course originator's experience with OOP and FP during a period of 30 years in industrial and academic software projects.

Introduction to Object Orientation
- Classes and Objects
- Attributes and behaviour
- Encapsulation and information hiding
- Constructors, setter and getter functions

Advanced Object Oriented Modelling
- Composition and inheritance
- Avoiding overuse of inheritance
- Polymorphism and Duck Typing
- Essential UML Class Diagrams

A gentle Introduction to Functional Programming

- What is functional programming?
- Function signature: 'pure' functional languages
- Lambda functions
- First-order and higher-order functions

Advanced Functional Programming

- A taxonomy of functions in Python
- Lazy and eager evaluation
- Recursion
- Currying and partial function evaluation

Modules and Packages

- Organising your classes
- Creating modules and accessing their contents
- Nested modules and packages
- Absolute and relative imports

Module 3 Object Oriented Programming in Python

In this module we show how to write *correct* object-oriented code in Python by which we mean code that is maintainable and that bears some resemblance to real-life applications. We transfer the reusable experience gained from our multi-year projects in C++ and C# (for example, Computer Aided Design (CAD), industrial software and computational finance) to Python. We focus on stripped-down but realistic examples that promote good design principles. In particular, we discuss the relative merits of inheritance and composition and when (and when not!) to use them. We also see how some famous *design patterns* naturally emerge when we design flexible software. In this way, we hope to get our designs right first time. More realistically, we wish to develop adaptable software that evolves as a series of prototypes. We quantify and formalise this remark in Module 6 when we introduce *Design Patterns*.

We also introduce a number of essential data types and containers that classes use to hold and share data.

Creating Classes in Python

- Naming conventions
- My first class A-Z
- Constructors and initialisation
- Creating objects and class instantiation
- Access control issues

Creating Larger Classes

- Composition and Delegation
- Whole-part objects in Python
- Arrays and collections of objects
- Inheritance and subclassing
- Combining inheritance and composition

Fundamental Arrays and Data Structures

- Basic data types
- One-dimensional arrays; matrices
- n-dimensional arrays (`ndarray`)
- Data type objects (`dtype`)
- Tuples, dictionaries and lists

Module 4 Functional Programming (FP) in Python

In this module we discuss the extent to which Python supports programming in a functional style. As such, Python is not a functional programming language. It does have a number of very useful features that we can use to promote the flexibility and maintainability of Python applications.

Functional programming decomposes a problem into a set of functions. Functions take input and produce output. Functions have no internal state that affects the output produced for a given input. It is

easy to compose function and to customise them to suit new requirements. It is also possible to define higher-order functions that accept functions as input and produce a function as output.

It is possible to combine the functional and object-oriented programming styles as we shall see in Module 6, in particular when creating *multiparadigm design patterns*.

Introduction to Higher-order Functions (HOFs)
- What can we do with HOFs?
- Simplify HOFs by lambda forms and lambda expressions
- Lambdas and the lambda calculus
- Apply a function to a collection: `map ()`
- Pass/reject data with `filter ()`

Advanced Functional Programming
- Lazy and eager evaluation
- Applications of `filter ()`
- Generator expressions
- Recursion and reduction
- Folds
- Decorators
- Iterables and *itertools* module

Exception Handling in Python Programs
- Raising exceptions
- Handling exceptions
- Exception hierarchy and built-in exceptions
- User-defined exceptions

Some Standard Library Modules
- *sys* (state of Python interpreter)
- *functools* (functions and types supporting functional programming)
- *heapq* (keep list "nearly sorted")
- *copy* (deep and shallow copies of objects)

Module 5 Advanced Python

In this module we introduce two major areas in the Python language. First, we discuss functionality and modules that process data and text, in particular file I/O, text I/O, serialisation and deserialisation of data and persistence. We also give an introduction to structured text and XML (*eXtensible Markup Language*).

The second part of this module is concerned with the application of Python functionality that uses the processing power of modern multiprocessor/multicore computers. The overriding reason for using this functionality lies in the code performance improvements when compared with code that runs on a single processor/core. To this end, we introduce three main techniques, namely *threads*, *processes* and *tasks* (*futures*). We also introduce the concepts of *concurrency* and *parallelism.*

Regular Expressions in Python
- What is a regular expression?
- The `re` module
- Pattern-string syntax
- Match and Search
- Regular expression objects

Files and File Operations
- The *io* module
- Creating file objects; attributes and methods
- In-memory Files
- Auxiliary Modules for File I/O
- Filesystem Operations

Text Input and Output
- Standard output and standard error
- Standard input
- Richer-text I/O
- Interactive command sessions

Structured Data: An Introduction to XML
- What is XML?
- *ElementTree* functionality and class
- *Element* class
- Parsing XML: the options

An Introduction to Numeric Processing
- The *math* and *cmath* modules
- Random and pseudorandom numbers

- The *array* module
- Global overview of *numpy*

Overview of Concurrency
- What is concurrency?
- Parallelism
- Threads and Tasks
- Processes

Threads in Python
- What is a thread?
- The *threading* Module
- Thread objects
- Thread synchronisation
- Locking
- Condition objects and event objects

Advanced Threading
- Timers
- Barriers
- Thread Local Storage
- The queue Module

Multiprocessing in Python
- What is a process?
- Sharing state: `Value`, `Array` and `Manager`
- Multiprocessing pools
- Comparing threads and processes

Introduction to Parallelism
- Data and task parallelism
- Data dependency graph
- Futures and promises
- Tasks

Tasks in Python
- The *concurrent.futures* module
- Thread and process pool executors
- The `Future` class
- Module functions
- Exception classes

Asynchronous Operations: Foundations
- Event-driven programming
- Blocking and nonblocking code
- Callback-based async architectures
- Coroutine-based async architectures

Asynchronous Operations in Python
- The *asyncio* module
- Coroutines in asyncio
- Event loops
- Connections and server
- Tasks and Futures
- I/O multiplexing

Persistence in Python (An Introduction)
- Serialization and deserialization
- Object persistence
- The *json* module
- The *pickle* and *cPickle* modules

DBM Modules
- Pre-relational databases
- Associative arrays
- Primary key and buckets
- Hash tables and extendible hashing
- Efficient high-speed storage

Module 6 Designing Maintainable Python Programs and Applications

In this module we introduce proven, unique (based on originator's 30 years of experience in software design) and repeatable strategies to design robust and maintainable Python code. We have created the topics in this module in order to fill a major gap (as we see it) in current programming practice. In very general terms, we take a top-down design approach by first defining what we wish to achieve (the goals of the software system) and then decomposing the system into loosely-coupled subsystems and modules that cooperate to satisfy those goals. In short, we take the following approach (to paraphrase the mathematician George Polya):

> *First, you have to understand the problem.*
> *After understanding, make a plan.*
> *Carry out the plan.*
> *Look back on your work. How could it be better?*

The basic approach is to keep refining our design until we arrive at the stage whereby we can map modules to Python language features and design patterns. To summarise, we use a combination of top-down decomposition and bottom-up programming and prototyping to ensure that we create the software product as demanded by the requirements.

A special and innovative feature of this module (based on the new ideas from the course originator) is how we upgrade and improve the somewhat outdated object-oriented design patterns from the 1990s by integrating them into a new multi-paradigm framework based on a flexible combination of the object-oriented and functional programming styles.

Software Architecture: Principles

- Context diagram and data flow
- System Decomposition; finding modules and classes
- Creating a software prototype and production systems
- Testing and debugging code

Design Best Practices

- Single Responsibility (SRP)
- Coupling and cohesion
- Information hiding
- Interface segregation
- Dependency inversion

An Introduction to Design Patterns

- What, why, when and how Design Patterns
- Creational, structural and behavioural patterns
- Discovering and using patterns in your applications
- The top design patterns

Creational Patterns

- Factory Method
- Abstract Factory
- Builder

Structural Patterns

- Adapter
- Façade
- Decorator
- Bridge

Behavioural Patterns

- Mediator
- Command
- Strategy and Template Method
- Visitor