Iterators and Generators in Python

- **Introduction**
  - What are Iterators and Generators?
  - What you will learn
  - Requirements

- **Understanding Iteration in Python**
  - Repating codes
  - Loops
    - While loop
    - For loop
      - Numeric range base
      - Three expression base
      - Iterator base

- **Iterables**
  - What is iterable?
  - Iterable protocol
    - `__iter__()` magic method

- **Iterators**
  - What is iterator?
  - Iterator protocol
    - `__iter__()` magic method
    - `__next__()` magic method
  - `StopIteration` exception
  - For-loop
    - Implementing a For-loop without using the `for` keyword
  - Built-in functions, `iter()` and `next()`

        - Review and analyze the source of these functions in *CPython*
        - Implementing these functions

- **Types of Iterators**
    - Classic
        - Linked-List iterator
    - Transformer
        - Square
        - Implementing a `map()` iterator
    - Generator
        - Fibonacci
        - Implementing `range()` as an iterator

- **Generators**
    - What is generator?
        - `yield` keyword
    - Types of generators
        - Classic
        - Transformer
        - Generator
    - Generator expression
        - Nested generators
    - Monitoring generators execution
        - `frame` objects
        - Generator properties
            - `gi_frame`
            - `gi_running`
            - `gi_suspended`

- **Generator Methods**
    - `send()` method

        - Reimplementing the `next()` function
        - Interactive `range()`
    - `throw()` method
        - Responsiveness
        - Error tracking
    - `close()` method
        - Resource management

- **Generator-based Coroutines**
    - Introduction to Execution Models
        - Sequential
        - Parallelism
            - Multi-Processing
        - Concurrency
            - Multi-Threading
            - Python GIL
            - Asynchronous Programming
    - `yield from` statement
    - Implementing coroutines
        - Generator-based coroutine
        - Implementing coroutines using `async/await`
            - `asyncio` Module
    - Async objects
        - AsyncIterator
        - AsyncGenerator
        - AsyncContectManager
            - `aiohttp` Module

- **Memory-Efficient Data Processing**
    - Iterators vs Containers

    - Pipelines

- **itertools Module**
    - Infinite Iterators
        - `count`
        - `cycle`
        - `repeat`
    - Terminating Iterators
        - `chain`
        - `accumulate`
        - `compress`
        - `filterfalse`
        - `dropwhile`
        - `takewhile`
        - `starmap`
        - `islice`
        - `batched`
        - `zip_longest`
        - `pairwise`
    - Combinatoric Iterators
        - `product`
        - `permutation`
        - `combinations`
        - `combinations_with_replacement`
    - itertools Recipes

- **Additionals**
    - Iterators constraints
        - Disposable
        - One-way iteration
        - No support for indexes and slices
    - Iterators vs Iterables

    - Reversible objects
        - Reversible protocol
            - `__reversed__()` magic method
        - Built-in `reversed` function
            - Implementing this function
    - more-itertools Module
        - Installation
        - Iterators
            - Grouping
            - Lookahead and lookback
            - Windowing
            - Augmenting
            - Combining
            - Summarizing
            - Selecting
            - Math
            - Combinatorics
            - Wrapping
            - Others

- **Conclusion**
    - References