[Page-12]

Iterators – Adv. Python concepts that allow for efficient looping & memory management.

→ It provide a way to access elements of a collection sequentially without exposing the underlying structure.

```
iterator = iter(my_list)
# to iterate through all element : next(iterator)   // Prints all elements one by one until last one then raise error
```

Generators: These are a simpler way to create iterators.

→ They use the yield keyword to produce a series of values lazily, which means they generate values on the fly & do not store them in memory.

→ Particularly useful for reading large files because they allow you to process one line at a time without loading the entire file into memory.

Decorators: Allows to modify the behaviour of a function or class method.

→ They are commonly used to add functionality to functions or methods without modifying their actual code.

SQL & Sqlite → self contained, serverless, and zero-config database engine that is widely used for embedded database systems.

Standard language for managing & manipulating relational databases.

```
import sqlite3

connection = sqlite3.connect('example.db')
connection

cursor = connection.cursor

cursor.execute('''
    # create table or insert values or update values or delete
''')

connection.commit()  # commit changes

cursor.execute('select * from employees')
rows = cursor.fetchall()

for row in rows:
    print(row)
```

Python Logging — provides a way to track events, and errors & operational information.

→ Python's built in logging module offers a flexible framework for emitting log messages from Python programs.

→ `import logging`

```
# configure basic logging settings   logging.basicConfig(level=logging.DEBUG)
# log messages
logging.debug("Message")
logging.info("info")
logging.warning("warning")
logging.error("!!")
logging.critical("!!")
```

→ logging module has several log levels indicating severity of events. Default levels are: DEBUG, INFO, WARNING, ERROR, CRITICAL

```
logger = logging.getLogger("module")
logger.setLevel(logging.Debug)
```

Memory Management with Python

→ Combination of automatic garbage collection, reference counting, and various internal optimizations to efficiently manage memory allocation & deallocation.

→ Reference counting: It is the primary method Python uses to manage memory. Each object in Python maintains a count of references pointing to it. When the reference count drops to zero, the memory occupied by the object is deallocated.

```
import sys
a = []
print(sys.getrefcount(a))
```

→ Garbage Collection: Python includes a cyclic garbage collector to handle reference cycles. Reference cycles occur when objects reference each other, preventing their reference counts from reaching zero.

```
import gc
```

```
gc.enable()
```

```
gc.disable()
```

```
gc.collect
```

etc...

→ Memory management best practices:

1) Use local variables
2) Avoid circular references
3) Use generators
4) Explicitly delete objects
5) Profile memory usage