PYTHON REVISION (Extended)

---

7. MODULES AND PACKAGES

1. Modules

A **module** is simply a Python file (`.py`) that contains code like functions, variables, or classes. Modules help in organizing and reusing code.

Importing Modules

```
import math
print(math.sqrt(16))  # Output: 4.0
```

Importing Specific Elements

```
from math import sqrt
print(sqrt(25))  # Output: 5.0
```

Renaming Modules

```
import math as m
print(m.pow(2, 3))  # Output: 8.0
```

Custom Module Example

greet.py

```
def say_hello(name):
    return f"Hello, {name}!"
```

main.py

```
import greet
print(greet.say_hello("Ajay"))
```

---

2. Packages

A **package** is a folder containing multiple module files and a special `__init__.py` file. This tells Python that the directory should be treated as a package.

Structure Example:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

Using a Package

```
from mypackage import module1
module1.function()
```

---

8. ERROR HANDLING (try, except)

Errors during program execution are called exceptions. Python provides `try`, `except`, `else`, and `finally` blocks to handle them.

Basic try-except

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print(result)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input.")
```

else and finally

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Not a number!")
else:
    print("Valid input.")
finally:
    print("Always runs.")
```

---

9. COMPREHENSIONS

Comprehensions provide a concise way to create lists, dictionaries, and sets.

1. List Comprehension

```
squares = [x**2 for x in range(5)]  # [0, 1, 4, 9, 16]
```

With Condition

```
even = [x for x in range(10) if x % 2 == 0]  # [0, 2, 4, 6, 8]
```

Nested List Comprehension

```
matrix = [[row * col for col in range(3)] for row in range(3)]
print(matrix)  # [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

2. Set Comprehension

```
unique = {x for x in [1, 2, 2, 3, 3]}  # {1, 2, 3}
```

3. Dictionary Comprehension

```
squares = {x: x*x for x in range(5)}  # {0:0, 1:1, 2:4, 3:9, 4:16}
```

---

10. ITERATORS AND GENERATORS

1. Iterators

An iterator is an object that allows you to iterate through all the elements of a collection using `__iter__()` and `__next__()` methods.

```
nums = [1, 2, 3]
it = iter(nums)
print(next(it))  # 1
print(next(it))  # 2
print(next(it))  # 3
```

You can create your own iterator by defining a class with `__iter__()` and `__next__()` methods.

```
class Count:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        self.current += 1
        return self.current - 1

for i in Count(1, 5):
    print(i)
```

2. Generators

Generators are a simpler way to create iterators using functions and the `yield` keyword.

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for num in count_up_to(3):
    print(num)
```

- **`yield`** pauses the function, saving its state for the next call.
- Generators are memory-efficient for large data sets.

You can also convert generators to lists:

```
gen = count_up_to(3)
print(list(gen))  # [1, 2, 3]
```