MODULE – 6 : PYTHON FUNDAMENTALS

1. Introduction to Python

Q1. Introduction to Python and Its Features

ANS: Python is a high-level, interpreted, and general-purpose programming language. It's widely used due to its simplicity and readability.

➢ **Features:**

- Simple & Readable Syntax – Easy to learn for beginners.
- Interpreted Language – Runs code line by line.
- High-Level Language – Abstracts low-level details.
- Cross-platform – Runs on Windows, macOS, and Linux.
- Dynamic Typing – No need to declare variable types.
- Large Standard Library – Comes with multiple useful modules.

Q2. History and Evolution of Python

ANS:

➢ **History and evolution:-**

- 1989 – Created by Guido van Rossum.
- 1991 – First version (Python 0.9.0) released.
- 2000 – Python 2.0 introduced list comprehensions, garbage collection.
- 2008 – Python 3.0 launched with better design (not backward-compatible).
- 2025 – Python 3.x is used in AI, ML, web development, automation, and many more areas.

Q3. Advantages of using Python over other programming languages.

ANS: Few advantages of python are as mentioned below:-

- Easy syntax and fast to develop.
- Strong community and huge library support.
- Used in various domains (web, AI, data science).
- Platform independent.
- Integration support with C/C++/Java.

Q4. Installing Python & Setting Up Development Environment

Ways to install Python:

1. Anaconda – Best for data science; includes Python, Jupyter, etc.
2. PyCharm – Full IDE for Python.
3. VS Code – Lightweight editor; install Python extension.

Q5. Writing and executing your first Python program.

Code:

```
print("Hello, world!")
```

2. Programming Style

Q6. Understanding Python's PEP 8 Guidelines.

ANS: PEP 8 is Python's official style guide. As per PEP 8, the following are the guidelines which must be followed while writing python code.

- Use 4 spaces per indent.
- Limit lines to 79 characters.
- Function/variable names must be snake_case instead of 'a' or 'user'.
- Class names must be 'CamelCase' instead of 'camel'
- Add whitespace around operators.

Q7. Indentation, comments, and naming conventions in Python.

- Indentation: Mandatory (4 spaces).
- Comments: Use # for single-line and """ for docstrings or multiple line comments.
- Naming: Use meaningful names, follow snake_case.

Q8. Writing Readable and Maintainable Code.

- Follow PEP 8.
- Write small, modular functions.
- Use docstrings and comments.
- Avoid deep nesting.
- Choose descriptive variable names.

3. Core Python Concepts

Q9. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

Data Types:

1. Numeric Types

- Integer (int): Whole numbers without decimals, e.g. 42, -7
- Floating-point (float): Numbers with a decimal part, e.g. 3.14, -2.5
- Both of these support arithmetic operations like +, -, *, /, etc.
- A sequence of characters enclosed in single, double, or triple quotes: "Hello", 'Python', """multi-line text""".
- Supports concatenation, splitting, slicing, and many built-in methods like .upper(), .split(), .replace(), and len(). Eg: "Hello"
- list: [1, 2, 3]
- tuple: (1, 2)
- dict: {'a': 1}
- set: {1, 2, 3}

Q10. Python Variables & Memory Allocation

- Python variables are references to objects.
- Memory is managed automatically using reference counting and garbage collection.

Q11. Python operators: arithmetic, comparison, logical, bitwise.

- Arithmetic Operator: +, -, *, /, %, //
- Comparison: ==, !=, >, <, >=, <=
- Logical: and, or, not
- Bitwise: &, |, ^, ~, <<, >>

4. Conditional Statements

Q12. Introduction to conditional statements: if, else, elif.

- if: Checks a condition. If it's True, executes a block of code.
- elif (else-if): Checks another condition if the first if was False.
- else: Runs if none of the above conditions are True.

CODE:

```
if x > 0:
    print("Positive number")
```

```
elif (x == 0):
    print("Zero")
else:
    print("Negative number")
```

Q13. Nested if-else conditions.

Example:

```
if (x > 0):
    if (x < 10):
        print("Small positive number")
    else:
        print("Large number")
```

5. Looping

Q14.Introduction to for and while loops.

- Loops let you repeat a block of code without typing the same lines over and over again. There are two main loop types in Python:
    - for loop
    - while loop

1. The for Loop

- Python's for loop is used to step through each element of an iterable(e.g. list, string, tuple, range, etc.)
- It works by *requesting an iterator* from the object, calling next() until it's done (StopIteration).

  Syntax:-

  ```
  for loop_variable in iterable:
          #block of code
  ```
- Examaple:-

  ```
  fruits = ["apple", "banana", "cherry"]
  for i in fruits:
      print("I like", fruit)
  ```

2. The while Loop:

- A while loop repeats **as long as a condition stays true**. It's ideal when the number of iterations is **not known in advance**.
- If the condition never changes, the loop becomes infinite unless stopped by a break.
- Syntax:-

```
while condition:
   # code if condition is True
else:
   # optional;
```

- Example:-

```
for i in range(5):
   print(i)
while x < 5:
   print(x)
   x += 1
```

Q15. How Loops Work in Python.

- for: Iterates over a sequence.
- while: Runs while a condition is True.

Q16. Explain Using Loops with Collections (lists, tuples , etc).

1. *for* Loop :-

- Example :-

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
   print(fruit)
```

2. *while* Loop with Indexing

- Example :-

```
numbers = (10, 20, 30)
i = 0
while i <  len (numbers):
   print(numbers[i])
   i += 1
```

3. Accessing Index and Value: enumerate()

```
names = ["Alice", "Bob", "Carol"]
```

```
for index, name in enumerate(names):
    print(f"{index}: {name}")
```

4. Looping with Conditions / Filtering

Combine loops with conditions to filter or process items :

- Example:-

```
ages = [12, 25, 30, 17, 40]
for age in ages:
    if age > = 18:
        print(f" Adult: {age}")
```

5. Nested Loops for Nested Collections

- Example:-

```
matrix = [[1, 2], [3, 4], [5, 6]]
    for row in matrix:
        for value in row:
            print(value)
```

6. Comprehensions

While not a loop statement, list comprehensions are a compact way to loop and build new collections:

- Examples:-

```
squares = [x**2 for x in range(5)]
```

6. Generators & Iterators

Q17. Understanding how generator works in Python.

- A generator function is any def that uses yield (instead of return). Calling it does not run the function , it returns a *generator object*, which implements the iterator protocol (__iter__ + __next__).
- Every generator is an iterator, but not every iterator is a generator. Generators automatically remember their state by the Python runtime; custom iterators must manage it manually.
- Compared to returning full lists:
  - They're lazy: values are produced *on demand*.
  - Use constant memory, even over huge or infinite sequences.
  - Allow premature exit (e.g. break) without computing the rest.

➢ Functions that yield values one at a time.

```
def gen():
    yield 1
    yield 2
```

Q18. Difference between yield and return.

- return: exit fast, compute all, lose state.
- yield: pause, remember state, produce gradually.

Q19. Understanding iterators and creating custom iterators.

➢ An **iterator** is an object that follows Python's *iterator protocol*, meaning it implements:

- __iter__(self): Returns the iterator object itself.
- __next__(self): Returns the next item in a sequence. Raises a StopIteration exception when there are no more items.

➢ **USAGE:-**

- Functions like iter(collection) convert an *iterable* (e.g. list, tuple, string) into an iterator.
- next(iterator) calls the iterator's __next__() method internally to fetch an element.
- A for-loop implicitly calls iter() once, then keeps calling next() until a StopIteration is raised, ending the loop.

➢ Objects with __iter__() and __next__() methods.

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

7. Functions & Methods

Q20. Defining and Calling Functions in Python

```
def greet(name):
   print("Hello", name)
greet("Alice")
```

Q21. Function Arguments

- Positional: def add(a, b)
- Keyword: add(a=1, b=2)
- Default: def add(a, b=5)

Q22. Scope of Variables in Python.

- Local: Inside functions.
- Global: Outside functions.
- Use global keyword to modify global variables.

Q23. Built-in Methods for strings, lists, etc.

1. String:

- All of these return new strings; the original string stays unchanged. Python offers many more like strip(), replace(), find(), join(), etc.
- Example:-

```
text = "Hello, PyThon!"

print(text.upper())         # "HELLO, PYTHON!"

print(text.lower())         #  "hello, python!"

print(text.title())         #  "Hello, Python!"

print(text.swapcase())    #  "hELLO, pYtHON!"
```

2. List Methods:-

- Common methods include: append(), clear(), copy(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort().
- Example:-

```
lst = ["apple", "banana", "apple"]
lst.append("cherry")     # adds at end
print(lst)              # → ['apple', 'banana', 'apple', 'cherry']

print(lst.count("apple"))
lst.remove("banana")
print(lst)

item = lst.pop(0)
print(item, lst)

lst.sort()
lst.reverse()
```

3. Dictionary Methods:

- As of Python 3.7+, insertion order is preserved by design. Key operations typically run in **O(1)** average time complexity.
  Some most usefull built-in methods are:- keys(), values(), get(), update(), copy(), pop().
  Example:-

```
my_dict = {'x': 10, 'y': 20}
```

```
d = my_dict.copy()
d.clear()

new_dict = my_dict.copy()
new_dict['a'] = 3
print(new_dict)

new = dict.fromkeys(['id', 'name', 'age'], None)

print(my_dict.get('x'))            # 10
print(my_dict.get('z', 0))         # 0 (default)

print(my_dict.keys(), my_dict values(), my_dict.items())

print(.pop('a'))           # 1
print(my_dict.pop('z', 'NA'))     # 'NA'

d = {'a' : 1, 'b' : 2, 'c' : 3}
print(d.popitem())             # ('c', 3)

print(d.setdefault('a', True))   # False (no change)

print(d.setdefault('b', '5'))  # '5'

# update()
d.update([‘d’,10])    #{‘d’:10}
```

4. Tuple Methods

- They offer just two non-special methods: .count() and .index(). Additional operations like slicing, concatenation, and built-in functions are supported through other language features.
- Although not methods, these built-in functions work well with tuples.

```
data = (5, 1, 9, 3)
len(data)        # 4
min(data)        # 1
max(data)        # 9
sum(data)        # 18
sorted(data)     # [1, 3, 5, 9]
```

- Example;-

```
t = (1, 2, 3, 2, 4, 2)

print(t.count(2))       # 3
print(t.index(3))       # 2

# Using index safely
```

```
if 5 in t:
   print(t.index(5))
else:
   print('5 not in tuple')
```

5. Other built-in functions:-
- These are global functions that typically accept not only lists but also other iterable types.
- **Example:-**

```
print(len(lst), sum([3,4,5]), max(lst))

print(sorted(lst))   # returns new sorted list

print("cherry" in lst)
```

8. Control Statements

Q24 Understanding the role of break, continue, and pass in Python loops.

1. Break:-
- break exits the loop right away—even if it's a nested loop (it only stops the innermost one).

Example:-

```
for n in range(1, 11):

   if n == 5:

      print("Breaking at", n)

      break

   print(n)
```

2. continue:-
- continue skips the rest of the current iteration and goes to the next. It doesn’t break out of the loop.

Example:-

```
for n in range(6):

   if n % 2 == 0:

      continue

   print(n)
```

3. pass:-
- pass is a no-op. It allows you to write syntactically valid code blocks without any action—useful for stubbing out logic while sketching code.

Example:-

```
for n in range(3):
   if n == 1:
       pass  # not implemented yet
   print(n)
```

9. String Manipulation

Q25. Understanding how to Access and Manipulate Strings

1. String Indexing — Accessing individual characters:

- Strings in Python are sequences indexed from 0 to len(s) – 1
- You can use negative indexes to count from the end (–1 is the last character).

2. String Slicing — Extracting substrings:

Slicing with s[start:stop:step] gives a new string:

- start is inclusive, stop is exclusive.
- Omitting start, stop, or step defaults to the start, end, or 1, respectively.
- Use negative step to reverse.

3. Immutability — Why strings won't change in place

- Strings in Python are immutable. Once created, you cannot modify individual characters or assignments like s[0] = 'J' will raise a TypeError.
- Immutability gives you safety (strings used as dictionary keys won't change unexpectedly) and enables optimization like string interning.

4. String manipulation methods — All return new strings

- Even methods that appear to modify the text (like .upper()) **do not change** the original.

Example:

```
s = "Python"
print(s[0])  # P
print(s[-1])  # n
text = "Hello, world!"
print(text[0:5])     # 'Hello'
print(text[7:])      # 'world!'
print(text[:5])        # 'Hello'
print(text[7:-1])    # 'world'
print(text[::2])      # 'Hlo ol!'
print(text[::-1])    # '!dlrow ,olleH'
s2 = s.upper()
print(s)              # "  Python  "
print(s2)             # "  PYTHON  "
#Trim whitespace
trim = s.strip()
print(trim)

# Search & replace
print("count of '2':", trim.count("2"))
print( trim.find("🐍")
swap = trim.replace("🐍", "snake")
print(swap)

#Split and rejoin
words = trim.split()
print(words)
joined = "|".join(words)
print(joined)

#Build a new modified string
modified = "→ " + swap[:10] + " … " + swap[-5:]
print("modified string:", modified)
```

Q26. Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

CODE:-

```
"Hello" + "World"   # Concatenation
"Hi" * 3                # Repetition
# String Methods:
s = "  PyThOn  "
print(s.upper())             # → "  PYTHON  "
print(s.lower())            # → "  python  "
print(s.capitalize())       # → "  python  "
print(s.title())            # → "  Python  "
print(s.swapcase())         # → "  pYtHoN  "

# Trimming & removing substring
t = "   apple banana apple   "
print("strip:", t.strip())                             # → "apple banana apple"
print("replace:", t.replace("apple", "orange"))        # → "   orange banana orange   "
print("count:", t.count("apple"))                      # → 2
print("find banana at index:", t.find("banana"))       # → 9 (or -1 if missing)
print("prefix apple?", t.strip().startswith("apple"))  # → True
print("suffix apple?", t.strip().endswith("apple"))    # → True

#Splitting & re-joining Strings
string = "one,two,three"
parts = string.split(",")         # → ['one', 'two', 'three']
rejoined = ";".join(parts)       # → "one;two;three"
```

Q27. String Slicing Example

```
s = "Python"
print(s[0:2])  # Py
print(s[::2])  # Pto
```

10. Advanced Python

Q28. How functional programming works in Python.

Here's how functional programming works in Python:-

1. Functions as First-Class Objects & Higher-Order Functions

Python treats functions like any other object: you can **assign** them to variables, **pass** them as arguments, **return** them from other functions, and **store** them in data structures such as lists or dictionaries—this qualifies Python to host higher-order programming styles. That's the essence of first-class functions in Python.

Example:

```
def twice(f, x):  /*
    return f(f(x))
def plus_three(n):
    return n + 3
print(twice(plus_three, 7))  # → 13
```

2. Anonymous Functions & Closures

- **Lambdas (Anonymous Functions)**

The lambda keyword defines a short, nameless function on the fly—perfect for use in map, filter, functional combinators, or closures. Since lambdas can only contain a *single expression*, they're concise by design.

```
square = lambda x: x * x
print(list(map(square, range(5))))  # => [0,1,4,9,16]
```

Closures

Python supports **lexical closures**, so inner functions can "remember" variables from enclosing scopes—even after that outer function has returned. This is commonly used in decorators and function factories.

```
def make_adder(n):
    def add(x):
        return x + n
    return add
add5 = make_adder(5)
print(add5(10))  # → 15
```

3. Functional Built-Ins: map(), filter(), reduce()

Python includes built-ins that embody functional paradigms:

- **map(f, seq)** applies f to each element, returning a lazy iterator (not a list).
- **filter(f, seq)** selects elements where f(x) is True.
- **reduce(f, seq)** cumulatively combines items; it resides in functools rather than built-ins.
- **Example:**

```
from functools import reduce
```

```
nums = [1,2,3,4,5]

evens = list(filter(lambda x: x % 2 == 0, nums))

squares = list(map(lambda x: x * x, nums))

product = reduce(lambda a, b: a * b, nums, 1)

print(evens)    # [2,4]

print(squares)  # [1,4,9,16,25]

print(product)  # 120
```

4. Comprehensions & Generator Expressions

Python's tightly integrated support for **list/set/dict comprehensions** and **generator expressions** embodies functional and declarative essence—expressing "what" instead of "how".

Example:

```
nums = range(1, 10)

squares = [x*x for x in nums]

evens   = {x for x in nums if x % 2 == 0}

stepals  = (2**n for n in range(1, 11))
```

Generators defer computation until needed, making them memory-efficient and side-effect free.

5. Immutability & Pure Functions

While Python allows mutable data, FP style encourages immutability:

- Use tuples/frozensets instead of lists/sets.
- For structured data, employ namedtuple, @dataclass(frozen=True), or use functional libraries like pyrsistent.
- Strive for **pure functions**: no side-effects, deterministic output, and no reliance on external mutable state.

6. Complementary Functional Modules

- **functools**: includes reduce, partial, lru_cache, and singledispatch. → Use partial(f, _args...) for partial application.
- **itertools**: for lazy, efficient iterator combinators (chain, cycle, accumulate, groupby, etc.).

- **operator**: includes functional versions of common operators—e.g. operator.add, useful in map/reduce, or sorting by key via operator.itemgetter.

These modules assist in building pipelines and compositions.

7. Decorators (Function Transformers)

A **decorator** is itself a higher-order function that **wraps or modifies** another function:

Example:

```
from functools import wraps

def memoize(f):
    cache = {}
    @wraps(f)
    def wrapper(*args):
        if args in cache:
            return cache[args]
        res = f(*args)
        cache[args] = res
        return res
    return wrapper


@memoize
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Decorators leverage closures and first-class functions to enhance behavior without altering function internals.

8. Recursion & Expression-centric Programming

Although Python doesn't optimize tail calls, you *can* write recursive algorithms mirroring functional languages:

Example:

```
def factorial(n):
```

```
return 1 if n == 0 else n * factorial(n-1)
```

Q29. Using map(), reduce(), and filter() functions for processing data.

1. **map():**

Syntax: map(func, iterable[, ...])

- Applies a function to each element of iterable(s), returning an iterator of transformed values.
- In Python 3, map() returns an iterator, not a list—so wrap with list() to inspect it
- Can accept multiple iterables, applying the function element-wise until the shortest iterable is exhausted.
- **Example:**

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16, 25]
```

2. **filter():**
   Syntax: filter(func, iterable)

- Keeps only those elements in the iterable for which func(element) returns True.
- Also returns an iterator, so use list() to materialize results.

- **Example:**

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4, 6]
```

3. **reduce():**
   Syntax: reduce(func, iterable[, initializer])

- Applies a binary function cumulatively to elements of the iterable to produce a single value.
- Not a built-in in Python 3 anymore—you must import it from functools.

Example:

```
from functools import reduce

result = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])

print(result)  # 15
```

Q30. Introduction to Closures and Decorators in Python.

Closure: A closure is a nested function that remembers variables from its enclosing scope, even after that outer function has finished executing.

- Define a function inside another function.
- The inner function uses a variable defined in the outer function.
- Return the inner function to the caller.

Example:

```
def outer(x):
    def inner():
        print(x)
    return inner
```

Decorator: A decorator is a callable that takes a function, modifies or wraps it, and returns a new function. Internally decorators are implemented using closures.

Example:

```
def decorator(func):
    def wrapper():
        print("Before")
        func()
        print("After")
    return wrapper


@decorator
def greet():
    print("Hello")
```
