Python – Collections, functions and Modules

(theory)

-Harsh Chauhan

1).Accessing List

1) Understanding how to create and access elements in a list.

- Creating lists in Python: Lists are made with square brackets [] separating items by commas, like fruits = ["apple", "banana", "cherry"]. You can also use list() constructor on strings or tuples, e.g., list("abc") gives ['a', 'b', 'c']. empty list is just [].

- Accessing by positive index: Index starts at 0, so fruits[0] is "apple", fruits[1] is "banana". Goes up to len(fruits)-1. Out of range gives IndexError.

- Negative indexing: Counts backward from end, fruits[-1] gets last item "cherry", fruits[-2] gets "banana". Super handy for end access without len()

- Slicing for ranges: fruits[1:3] gives ["banana", "cherry"] (start inclusive, end exclusive). fruits[:2] from start, fruits[2:] to end, fruits[:] copies whole list

- Index and value together: Use enumerate() in for loop: for i, fruit in enumerate(fruits): print(i, fruit) shows 0 apple, 1 banana etc. Perfect for needing both

2) Indexing in lists (positive and negative indexing).

- Positive Indexing: Starts at 0 for the first item, goes up like list for first, list for second, up to list[n-1] for the last (where n is list length).
- Negative Indexing: Starts at -1 for the last item, counts backwards like list[-1] last, list[-2] second-last, super handy for end access without knowing length.
- Example: For mylist =, mylist is 20 (positive), mylist[-2] is 30 (negative).

Positive for front, negative for back—makes code shorter, like popping last with pop(-1).

3). Slicing a list: accessing a range of elements.

- Slicing Syntax: Use list[start:end:step] to grab a range—start inclusive, end exclusive, step skips elements (defaults: start=0, end=last, step=1).

- Basic Example: For nums =, nums[1:4] gives; nums[:3] gets first three; nums[2:] takes from third to end.

- With Step: nums[::2] every second from start; nums[::-1] reverses whole list; nums[1:5:2] skips one.

2. List Operations

1) Common list operations: concatenation, repetition, membership.

- Concatenation: Join lists with + operator, like list1 + list2 makes a new list; or use extend() to add to existing one (list1.extend(list2)).

- Repetition: Multiply list by number, e.g., fruits = ['apple'] * 3 gives ['apple', 'apple', 'apple'] repeats elements.

- Membership: Check if item in list with in keyword, like if 'banana' in fruits: True/False, fast for small lists.

2) Understanding list methods like append(), insert(), remove(), pop().

- append(): Adds one item to the end of list, like fruits.append('mango')—list grows by one.

- insert(): Puts item at specific spot, e.g., fruits.insert(1, 'kiwi') shifts rest right.

- remove(): Deletes first match of value, fruits.remove('apple')—needs exact match or error.

- pop(): Removes/returns item at index (default last), x = nums.pop(2) gives value and shrinks list.

3. Working with Lists

1) Iterating over a list using loops.

- Basic For Loop: Simplest way for item in mylist: loops through each element one by one. Example: fruits = ['apple', 'banana', 'cherry']; for fruit in fruits: print(fruit) outputs apple, then banana, then cherry. No need for indices here, just grabs values directly.

- Loop with enumerate(): When you need both index and value, use enumerate(mylist). Like: for index, fruit in enumerate(fruits): print(f"{index}: {fruit}") gives 0: apple, 1: banana, etc. Starts index at 0 by default, or enumerate(fruits, start=1) for 1-based. Super useful for lists needing position info

- Using range(len()): For index-only access or modifying list: for i in range(len(mylist)): then do mylist[i]. Example: numbers =; for i in range(len(numbers)): numbers[i] *= 2 prints doubled values. Good when you gotta change elements during a loop.

- While Loop Alternative: Less common for lists, but i=0; while i < len(mylist): print(mylist[i]); i+=1. Stick to for loops though—they're cleaner and less error-prone for iteration.

2). Sorting and reversing a list using sort(), sorted(), and reverse().

- sort() method: Sorts list in place (changes original), like nums =; nums.sort() makes. Use sort(reverse=True) for descending:. No return value, just modifies list.

- sorted() function: Returns new sorted list, original stays same. sorted(nums) gives copy; sorted(nums, reverse=True) for descending. Works on any iterable.

- reverse() method: Reverses order in place, nums.reverse() turns to. No sorting, just flips.

- reverse vs reversed(): reverse() changes original; reversed(nums) gives iterator for new reversed copy, like list(reversed(nums)).

- Examples: fruits = ['banana', 'apple']; fruits.sort() → ['apple', 'banana']; sorted_fruits = sorted(fruits, reverse=True) → ['banana', 'apple']

3). Basic list manipulations: addition, deletion, updating, and slicing.

- Addition: Use append(item) to add at end, insert(index, item) for specific spot, or extend(list) to add multiple—like nums =; nums.append(3) →; nums.insert(0,0) →

- Deletion: remove(value) deletes first match, pop(index) removes/returns item (default last), del list[index] for any position.e.g., nums.remove(2) gone; x = nums.pop() gets last.

- Updating: Change by index, nums = 10 replaces second item directly.

- Slicing for Manipulation: Slice assigns ranges too, like nums[1:3] = replaces slice; del nums[1:3] deletes range; nums[2:2] = inserts without replacing

4. Tuple

1). Introduction to tuples, immutability.

- What is a Tuple?: Tuple is like a list but uses parentheses () instead of [], stores multiple items in order, e.g., coords = (10, 20, 30). Great for fixed data like points or records.

- Immutability Key Point: Once created, you can't change, add, or delete items, no append(), no = 5. Try it and get TypeError. Safer for data that shouldn't mess up, like constants or dict keys.

- Creating Tuples: Single item needs comma: t = (5,); empty is (). Mix types: person = ('Alice', 25, True). Use tuple() constructor too: tuple().

- Why Use Over Lists?: Faster, uses less memory, hashable (for dicts/sets). Still slice/index like lists: coords is 20, coords[1:3] is (20,30).

- Example: fruits = ('apple', 'banana'); print(fruits[-1]) → 'banana'.

2). Creating and accessing elements in a tuple.

- Creating Tuples: Use parentheses with commas: my_tuple = (10, 20, 'hi') or without parens a, b, c = 1, 2, 3. A single item needs a comma: (5,) not (5). Empty: (). Constructor: tuple([1,2,3]).

- Accessing Elements: Same as lists—positive index my_tuple[0] gets first (10), negative my_tuple[-1] gets last ('hi'). Slicing too: my_tuple[1:3] → (20, 'hi'). No changes since immutable

- Examples:
  - person = ('Alice', 25); print(person[0]) → 'Alice'
  - nums = (1,2,3,4); print(nums[-2:]) → (3,4)
  - colors = ('red',); print(colors) → ('red',)

3) Basic operations with tuples: concatenation, repetition, membership.

- Concatenation: Use + operator to join tuples into new one, like t1 = (1,2); t2 = (3,4); combined = t1 + t2 → (1,2,3,4). Creates fresh tuple, original unchanged.

- Repetition: Multiply with *, e.g., colors = ('red',); repeated = colors * 3 → ('red','red','red'). Handy for patterns, but watch memory for big repeats.

- Membership: in keyword checks existence, if 'blue' in colors: True/False. Fast lookup, same as lists—e.g., 2 in (1,2,3) → True.

- Examples:
  - nums = (10,20) + (30,) → (10,20,30)
  - twice = (5,) * 2 → (5,5)
  - 'apple' in ('apple','banana') → True
- All return new tuples since immutable—no in-place changes like lists.

5. Accessing Tuples

1). Accessing tuple elements using positive and negative indexing.

- Positive Indexing in Tuples: Starts at 0 for first item, just like lists.e.g., coords = (10, 20, 30); coords gets 10, coords gets 20. Goes up to coords for last.

- Negative Indexing: Counts from end with -1 as last,coords[-1] is 30, coords[-2] is 20, coords[-3] is 10. Perfect for grabbing end without len().

- Examples:
  - fruits = ('apple', 'banana', 'cherry'); fruits → 'banana' (positive).
  - fruits[-1] → 'cherry' (negative, last one).
  - fruits[0:2] → ('apple', 'banana'); fruits[-2:] → ('banana', 'cherry').
- Same slicing/index rules as lists, but read-only since immutable

2). Slicing a tuple to access ranges of elements.

- Tuple Slicing Syntax: Same as lists,tuple[start:end:step], start inclusive, end exclusive, step optional (default 1). Returns new tuple, original unchanged due to immutability.

- Basic Slicing Examples: For nums = (0,1,2,3,4,5):
    - nums[2:5] → (2,3,4) (from index 2 to before 5).
    - nums[:3] → (0,1,2) (start to before 3).
    - nums[3:] → (3,4,5) (from 3 to end).

- With Step: nums[::2] → (0,2,4) every second; nums[1:5:2] → (1,3) skips one; nums[::-1] → (5,4,3,2,1,0) reverses whole thing.

- Negative Indices: nums[-3:] → (3,4,5) last three; nums[:-3] → (0,1,2) first few; nums[-4:-1] → (2,3,4). Mix positive/negative works too.

6. Dictionaries

1). Introduction to dictionaries: key-value pairs.

- What is a Dictionary?: Dict stores data as key-value pairs using curly braces {}, like person = {'name': 'Alice', 'age': 25}.keys unique, values anything (int, str, list, etc.). Unordered (pre-Python 3.7), now insertion order kept.

- Key-Value Basics: Key (left of colon) must be immutable (str, int, tuple), value (right) flexible. No duplicate keys,later overwrites earlier. Access value: person['name'] → 'Alice'.

- Creating Dicts: Empty: d = {}; from pairs: d = {'a':1, 'b':2}; constructor: dict(a=1, b=2) or dict([('a',1)]). Great for lookups like phonebooks.

- Why Use Dicts?: Fast lookups (O(1) avg), flexible,no fixed size like lists. Mix types: inventory = {'apple': 10, 'banana': [1,2]}. Perfect for configs or data records in assignments.

- Example: student = {'id': 101, 'grades': [85,90]}; print(student['id']) → 101. KeyError if missing key use get('key', 'default') safely

2). Accessing, adding, updating, and deleting dictionary elements.

- Accessing Elements: Use key in brackets like person['name'] → 'Alice'. Missing key? KeyError. Safer: person.get('age', 'N/A') returns value or default.

- Adding Elements: Assign new key: person['city'] = 'NYC' adds it. Or update() for multiple: person.update({'job': 'dev'}). Keys auto-unique.

- Updating Elements: Reassign existing key: person['age'] = 26 changes value. update() works too for bulk: overwrites if key exists.

- Deleting Elements: del person['city'] removes key-value. pop('age') removes and returns value (26). clear() empties whole dict.

- Examples:
  - Start: d = {'a':1}
  - Add: d['b'] = 2 → {'a':1, 'b':2}
  - Update: d['a'] = 10
  - Delete: del d['b'] → {'a':10}

3) Dictionary methods like keys(), values(), and items().

- keys() method: Returns view of all keys, like person.keys() → dict_keys(['name', 'age']). Loop: for k in person.keys(): print(k) lists 'name', 'age'. Dynamic—updates if dict changes.
- values() method: Gets view of values only, person.values() → dict_values(['Alice', 25]). Iterate: for v in person.values(): print(v) shows 'Alice', 25. Handy for value checks.
- items() method: Returns key-value pairs as tuples, person.items() → dict_items([('name', 'Alice'), ('age', 25)]). Best for loops: for key, value in person.items(): print(key, value).
- Examples:
  - d = {'a':1, 'b':2}; print(list(d.keys())) → ['a', 'b']
  - print(list(d.values())) → [1, 2]
  - for k,v in d.items(): print(k,v) → a 1, b 2

7. Working with Dictionaries

1). Iterating over a dictionary using loops.

- Default Iteration (Keys Only): for key in mydict: loops over keys automatically. Example: person = {'name':'Alice', 'age':25}; for k in person: print(k) → name, age. Get value inside: print(person[k]).

- Iterate Values: for value in person.values(): just values prints Alice, 25. Clean when you don't need keys.

- Key-Value Pairs: for key, value in person.items(): unpacks tuples and prints name Alice, age 25. Best for most cases, super readable.

- With Index: for i, (k, v) in enumerate(person.items()): adds numbers: 0 name Alice, etc. Or enumerate(person) for keys only.

- Examples:
  - grades = {'math':90, 'sci':85}; for sub, score in grades.items(): print(f"{sub}: {score}")
  - Sorted: for k in sorted(grades): print(k, grades[k]) → math 90, sci 85.
- items() fastest for pairs. Dict views update live if dict changes mid-loop (rare).

2). Merging two lists into a dictionary using loops or zip().

- Using zip() (Easiest): zip(keys_list, values_list) pairs them into tuples, then dict() makes a dictionary. Example: keys = ['name', 'age']; values = ['Alice', 25]; person = dict(zip(keys, values)) → {'name':'Alice', 'age':25}. Handles unequal lengths by stopping at shorter.

- Loop Method: Iterate and assign: person = {}; for k, v in zip(keys, values): person[k] = v. Same result, more control—like skipping or checking duplicates.

- Dict Comprehension: Fancy one-liner: {k: v for k, v in zip(keys, values)}. Pythonic, fast for assignments.

- Examples:
    - fruits = ['apple', 'banana']; prices = [10, 20]; inventory = dict(zip(fruits, prices)) → {'apple':10, 'banana':20}
    - Unequal: zip(['a','b'], [1,2,3]) → {'a':1, 'b':2}

3). Counting occurrences of characters in a string using dictionaries.

- Use a dictionary where each character is a key and its count is the value.
- Start with an empty dictionary:
  freq = {}
- Loop over each character in the string:
  for ch in s:
- Inside the loop, update the count:
  - If character already present, increase its value
  - If not present, add it with value 1
- Typical pattern:

```python
s = "aabbaba"

freq = {}

for ch in s:

    if ch in freq:

        freq[ch] += 1

    else:

        freq[ch] = 1

print(freq)    # {'a': 4, 'b': 3}
```

- Dictionary :
  - Keys → characters (like 'a', 'b', ' ')
  - Values → how many times they appear in the string.

8. Functions

1). Defining functions in Python.

- Basic Syntax: Use def keyword + function name + parentheses + colon, then indent body. Like:

```python
def greet():

    print("Hello!")
```

- Call with greet() runs the code inside.
  With Parameters: Add vars in () for inputs:

```python
def add(a, b):

    return a + b

result = add(5, 3)   # 8
```

- Parameters get values when called. Return optional, sends back data.

- No Parameters Example:

```
def menu():

    print("1. Add 2. Subtract")

menu()   # Shows menu
```

- Key Rules:
  - Name like variables (no spaces, lowercase).
  - Indent body (4 spaces).
  - Call anywhere after definition.
  - return ends function, gives output (or None if missing).

2). Different types of functions: with/without parameters, with/without return values.

- No Parameters, No Return: does action like print.

```
def greet():
    print("Hello!")
greet()  # Just prints, returns None
```

- With Parameters, No Return: Takes inputs, does something (side effect).

```
def add_print(a, b):
    print(a + b)  # Prints 8
add_print(3, 5)
```

- No Parameters, With Return: Fixed calc, returns value.

```
def get_pi():
    return 3.14
pi = get_pi()  # pi = 3.14
```

- With Parameters, With Return: Most common—inputs in, result out.

```python
def multiply(x, y):
    return x * y
result = multiply(4, 5)  # 20
```

3). Anonymous functions (lambda functions).

- Lambda Basics: Anonymous one-liner functions using lambda args: expression → no def, no name needed. Super short for simple ops. Example: double = lambda x: x*2; double(5) → 10.

- Syntax Breakdown: lambda param1, param2: param1 + param2. Multiple args ok, but only one expression (no statements/multi-lines). Returns expression results automatically.

- Common Uses:
  - With map(): list(map(lambda x: x**2, [1,2,3])) → [1,4,9]
  - With filter(): list(filter(lambda x: x>2, [1,3,2,4])) → [3,4]
  - Sorting: sorted(['banana','apple'], key=lambda x: len(x)) → ['apple','banana']

- Multiple Args Example: add = lambda a,b: a+b; add(3,4) → 7. Default args too: lambda x, y=10: x+y.

- When to Use: Quick callbacks for map/filter/sorted, not complex logic (use def instead). Closures work: def maker(n): return lambda x: x*n.

9. Modules

1) Introduction to Python modules and importing modules.

- What are Python Modules?: Modules are .py files containing reusable code like functions, classes, variables → think code libraries to avoid rewriting stuff. Built-in ones like math, random come with Python; custom ones you create.

- Why Use Modules? Organizes big programs, reuses code across files, keeps things clean.

- Importing Basics:
  - Full module: import math; math.sqrt(16) → 4.0
  - Specific: from math import sqrt; sqrt(16) → cleaner
  - Alias: import math as m; m.pi → shorter names

- Creating Your Own:

```
# mymath.py
def add(a, b):
    return a + b
PI = 3.14
```

- Then: import mymath; print(mymath.add(2,3)) → 5

2) Standard library modules: math, random.

- math Module: Built-in for math ops—import with import math. Key functions: math.sqrt(16) → 4.0, math.pi → 3.14159, math.factorial(5) → 120, math.ceil(3.7) → 4, math.floor(3.7) → 3.

- random Module: For random numbers/values → import random. Essentials: random.randint(1,10) → random int 1-10, random.choice(['a','b','c']) → picks one, random.random() → 0.0-1.0 float, random.shuffle(list) mixes in place.

- Examples:
  - Math: import math; print(math.pow(2,3)) → 8.0
  - Random: import random; nums = [1,2,3]; random.shuffle(nums); print(nums) → shuffled like

- Usage: Always import first. Math for calcs/angles, random for games/simulations. No params needed for constants like math.e.

3). Creating custom modules.

- What is a Custom Module?: Just a .py file with your functions/classes/variables. Save as mymodule.py, then import anywhere. Reuses code across projects

- Step-by-Step Creation:
  - Create calc.py file:

```python
def add(a, b):
    return a + b

def multiply(x, y):
    return x * y

PI = 3.14
```

  - In another file main.py (same folder): import calc
  - Use: print(calc.add(5, 3)) → 8; print(calc.PI) → 3.14.

- Import Options:
  - import calc → use calc.function()
  - from calc import add → direct add(5,3)
  - from calc import * → all functions direct (avoid in big projects)

- Example Folder Structure:

```
myproject/
├── main.py
└── calc.py       # Your custom module
```