Python Basics

Tuesday, 12 November 2024 8:47 PM

Key tips:

- No Semicolons and Curly Braces but they use indentation (usually 4 spaces) to define code blocks.
- No need to declare variable types explicitly
- Python typically uses snake_case (e.g., my_variable) instead of camelCase
- Python doesn't use public, private, or protected modifiers like Java. Instead, it uses a naming convention
  - Prefixing a variable with a single underscore _variable suggests it is for internal use.
  - Prefixing with double underscores __variable invokes name mangling for class-specific attributes.
- **String Formatting**: Instead of Java's + operator for concatenation or String.format, Python has several ways to format strings:
  - **f-strings**: (Python 3.6+): name = "Alice"; print(f"Hello, {name}")
  - **format() method**: print("Hello, {}".format(name))
  - **Multi-line Strings**: Use triple quotes (''' or """) for multi-line strings. E.g message = """This is a multi-line string in Python."""
- **Lists** in Python are similar to Java's ArrayList but are dynamically sized and can store mixed data types
- **Dictionaries** in Python are like Map in Java, allowing key-value pairs
- **Sets** work similarly in both languages, but Python makes set operations (like union, intersection) more convenient.
- **Functions and Methods**
  - **Defining Functions**: Python uses def instead of specifying return types, and functions return None if there's no return statement.
  - In Python, instance methods explicitly declare self as the first parameter to refer to the object itself (similar to this in Java).

    -
      ```
      class Dog:
          def __init__(self, name):
              self.name = name
      ```
- **Classes:** In both Java and Python, **classes** are blueprints for creating objects, encapsulating data (attributes) and behavior (methods). Defining classes in Python is simpler. The __init__ method acts as the constructor, and inheritance doesn't require extends. The constructor method in Python is always named __init__, and you don't need the new keyword to create an object.
- **Interfaces and "Duck Typing" in Python:** In Python, **interfaces** aren't required. Instead, Python uses **duck typing**, meaning if an object has the required methods or properties, it can be used in that context, regardless of its class.
- **Abstract Base Classes as Python's Interface Substitute:** If you want to enforce that classes implement certain methods (similar to Java interfaces), you can use **Abstract Base Classes (ABCs)** in Python.
- **Exception Handling:** Python's try/except is similar to Java's try/catch. However, Python doesn't require specific exception types in the method signature.
- **Libraries and Imports:** Python's import system allows you to import modules or specific functions from them. You don't need package declarations at the beginning of files.
  - Python's standard library is extensive and contains modules for many common tasks (e.g., math, datetime, json, etc.), often replacing the need for additional libraries.
- **Variable References:** Keep in mind that variable assignments create references, not copies. Use .copy() or list() to explicitly copy mutable collections.
- **Running Code:**
  - **Running Scripts**: In Python, you can run a script directly from the command line using python script.py. There's no need for compilation (javac), as Python is interpreted.
  - **Interactive Mode**: Python has an interactive mode (python in the terminal), allowing you to test code snippets interactively.

	Java	Python

If conditions	if (a > b) {
	System.out.println("a is greater");
	}
		if a > b:
		    print("a is greater")

Data types	int number = 10;	number = 10

List	List = new ArrayList<Integer>();	my_list = [1, "two", 3.0]
		
		`res.append(one)`

Map vs Dict	Map = new HashMap<String, Int>();sort	my_dict = {"key1": "value1", "key2": "value2"}
		empty_dict = {}
		# getOrDefault
		dict.get(key, default_value)
		# put into map pull and pullAll
		my_dict.update({'c': 3, 'd': 4}) /
		My_dict[c] = 3
		# iterate and get key and value
		for i, j in water.items():
		    print(f"Key (i): {i}, Value (j): {j}")
		for key in my_dict:
		    print(key, "->", my_dict[key])
		# add if present, else add 1
		`m = {}`
		`for num in nums:`
		`    if num in m:`
		`        m[num] += 1`

		else:
		        m[num] = 1
		
		# delete from map:
		del dict[key] or dict.pop(key)
		
		Setting default value as list and adding to list:
		graph.setdefault(u, []).append(v)

Sets	Set = new HashSet<Integer>();	my_set = {1, 2, 3}
		empty_set = set()
		Empty_set.add(1)
		# to check whether its in set
		If 1 in empty_set:
		# Adding array into set:
		        Pairs = set()
		        pairs.add((arr[i], arr[j]))

Functions and Methods	Public String greet(String name){}	def greet(name): return f"Hello, {name}"

Classes	public class Dog {
	    String name;
	
	    public Dog(String name) {
	        this.name = name;
	    }
	
	    public void bark() {
	        System.out.println("Woof!");
	    }
	}
	
	// Create a Dog object in Java
	Dog myDog = new Dog("Buddy");
	myDog.bark();
		class Dog:
		    def __init__(self, name): # Constructor method
		        self.name = name
		
		    def bark(self):
		        print("Woof!")
		
		# Create a Dog object in Python
		my_dog = Dog("Buddy")
		my_dog.bark()

Interface	interface Animal {
	    void makeSound(); // Any class implementing Animal must have this method
	}
	
	public class Dog implements Animal {
	    public void makeSound() {
	        System.out.println("Woof!");
	    }
	}
		class Dog:
		    def make_sound(self):
		        print("Woof!")
		
		class Cat:
		    def make_sound(self):
		        print("Meow!")
		
		# A function that expects any object with a \`make_sound\` method
		def animal_sound(animal):
		    animal.make_sound()
		
		# Works with both Dog and Cat, as they both have \`make_sound\`
		dog = Dog()
		cat = Cat()
		
		animal_sound(dog) # Output: Woof!
		animal_sound(cat) # Output: Meow!

Abstract class as interface		from abc import ABC, abstractmethod
		
		class Animal(ABC): # Abstract base class
		    @abstractmethod
		    def make_sound(self):
		        pass
		
		class Dog(Animal):
		    def make_sound(self):
		        print("Woof!")
		
		# Trying to instantiate Animal directly will raise an error
		# animal = Animal() # Error
		
		# Correct usage with subclass
		dog = Dog()
		dog.make_sound() # Output: Woof!

Exception Handling		try:
		    risky_operation()
		except Exception as e:
		    print(f"An error occurred: {e}")

Package and Import	Package com.abc.file; Import java.util.*;	import math from datetime import datetime

Practical Comparisons:

	Java	**Python**
Main method		if __name__ == "__main__":
Operators	&&	and
increment	++	Inc+=1
If condition ':'		if one < two and one < costs[0] **:**
Else if		elif
Access last element	days[days.length - 1]	days[-1]
For each		for day in days:
Initialize boolean array to false		travel = [False] * length
For i=0 to len and len to 0		for i in range(length): `for i in range(ln - 1, 0,-1):`

If(!value) & contains		if not value:
		vowels.__contains__(start)

Priority Queue	PriorityQueue<int[]> pq =
	new PriorityQueue<int[]>((a,b)->b[0]-a[0]);
	for(int i=0; i<arr.length; i++){
	pq.add(new int[]{arr[i], i})
	}
		import heapq
		priority_queue = []
		heapq.heappush(priority_queue, 5)
		heapq.heappop(priority_queue)
		
		# pq with int[]
		pq = []
		for i in range(len(arr)):
		heapq.heappush(pq, (-arr[i], i)). # - is for max heap
		
		Retrive using heappop

String Builder		parts = []
		for i in range(5):
		        parts.append(i)
		result = "".join(parts)

String char	S.charAt(2) S.charAr(i)-'0'	s[2] int(s[2])
Using variables inside string		parts.append(f"Number: {i}")

		min(a, b),
		  max(a, b),
		  math.pow(a, b),
		  math.sqrt(a),
		  abs(a)

Not Empty	If(!pq.isEmpty())	If pq:
Not null		is not None

String operations		my_string = "hello"
		string_length = len(my_string)
		for char in my_string:
		     print(char)
		# for getting substring of a string
		string[start:end]
		# Reversing a string
		reversed_s = ''.join(reversed(s)) or reversed_s = s[::-1]
		`reverse = words[i][::-1]`
		
		# concat
		result = s1 + " " + s2
		# upper and lower and camel case
		s.upper() & s.lower() & s.title()
		# remove white space, left extra space or right extra space
		s.strip() & s.lstrip() & s.rstrip()

		# splitting and joining
		s.split() & " ".join(words)
		# find and replace
		s.find("world") & s.replace("world", "Python")
		# starts with and ends with
		s.startswith("hello") & s.endswith("world")
		# alphanum, alpha and digit
		s.isalnum() & s.isdigit() & s.isalpha()
		# replacing immutable string (convert to list and join)
		res = list(res)
		res[i] = '1'
		res = ''.join(res)
		#Sub string of a string:
		for i in range(len(s) + 1):
		  new_str = s[:i] + ch + s[i:]

Type cast char to int		Str = 1101202132 -> int(ch)
		Str = "abcd" -> freq[ord(ch) - ord('a')]

Reversing a list		right.reverse()

Arrays sorting using different methods	Arrays.sort(words)	# sort based on the length of the word
		Words.sort(key=len)
		# Sort alphabetically by the last character of each word:
		words.sort(key=lambda word: word[-1])
		# Sort by length in descending order
		words.sort(key=len, reverse=True)
		# Sort alphabetically ignoring case
		words.sort(key=str.lower)
		# Sort by length first, then alphabetically
		words.sort(key=lambda word: (len(word), word))
		# Sorting Numeric Strings
		words.sort(key=int)
		# Soring list like Arrays.sort
		meetings.sort(key=lambda x: x[0])
		# If you have a mapping for each word, use it in the key
		priority = {"banana": 2, "apple": 1, "cherry": 3, "kiwi": 4, "fig": 5}
		words.sort(key=lambda word: priority[word])
		# initializing array of size
		alpha = [0] * 26
		# Initialize a list of lists for 26 characters
		char_indices = [[] for _ in range(26)]
		# 2D Array in Python
		dp = [[0] * 26 for _ in range(26)]
		
		#Reverse sort array:
		nums = sorted(nums, reverse=True)

Function's		#calling a function in same class
		Self.getfreq()
		#defining a function
		def getfreq(self, word: str) -> dict:

Scanner to get input	scanner.nextLine()	Input() -> String input int(input()) -> int input Float(input()) -> float input map(int, input().split()) -> multiple input

Creating objects and using them		class CustomObject:
		    def __init__(self, position, value, negatives):
		        self.position = position # Integer representing position
		        self.value = value # Any data type for value
		        self.negatives = negatives # List of integers (negative values expected)
		
		    def __repr__(self):
		        # Optional: Representation for easy debugging
		        return f"CustomObject(position={self.position}, value={self.value}, negatives={self.negatives})"
		
		# Create a list of CustomObject instances
		objects = [
		    CustomObject(1, 100, [-1, -2, -3]),
		    CustomObject(2, 200, [-10, -20]),
		    CustomObject(3, 300, [-5, -15, -25])

		]
		
		# Access and print the list of objects
		for obj in objects:
		    print(obj)
		
		# Access attributes
		print("First object's negatives:", objects[0].negatives)
		print("Second object's value:", objects[1].value)

Queue and Stack		#Queue
		# Initialize a queue
		queue = deque([[i, j]])
		while queue:
		# Remove an element from the front
		        current = queue.popleft()
		        print(f"Processing: {current}")
		# Add elements to the back of the queue (if needed)
		queue.append([current[0] + 1, current[1]])
		for element in queue:
		        print(f"Queue element: {element}")
		
		#Stack
		stack = [[i, j]]
		while stack:
		# Remove an element from the top
		        current = stack.pop()
		        print(f"Processing: {current}")
		# Add elements to the top of the stack (if needed)
		        stack.append([current[0] - 1, current[1]])
		for element in stack:
		        print(f"Stack element: {element}")

Object declaration POJO		class Group:
		    def __init__(self, low, high, values, parent):
		        self.low = low
		        self.high = high
		        self.values = values
		        self.position = 0
		        self.parent = parent

TreeMap and getting celing		from typing import List
		from collections import defaultdict
		import bisect
		
		class Solution:
		    def lexicographicallySmallestArray(self, nums: List[int], limit: int) -> List[int]:
		        sorted_nums = sorted(nums)
		        groups = []
		        res = [-1] * len(sorted_nums)
		        map_ = {} # Dictionary to mimic TreeMap behavior
		        group_index = 0
		
		        # Create groups
		        i = 0
		        while i < len(sorted_nums):
		            g = Group(low=sorted_nums[i], high=0, values=[], parent=0)
		            g.values.append(sorted_nums[i])
		
		            # Group numbers within the limit
		            while i + 1 < len(sorted_nums) and sorted_nums[i + 1] - sorted_nums[i] <= limit:
		                g.values.append(sorted_nums[i + 1])
		                i += 1
		
		            g.high = sorted_nums[i]
		            g.parent = g.high
		            groups.append(g)
		            map_[g.parent] = group_index
		            group_index += 1
		            i += 1
		
		        # Construct the result array
		        for i, val in enumerate(nums):

		# Find the ceiling key
		keys = sorted(map_.keys())
		celi_idx = bisect.bisect_left(keys, val)
		celi = keys[celi_idx]
		
		# Get the corresponding group and assign the value
		idx = map_[celi]
		group = groups[idx]
		res[i] = group.values[group.position]
		group.position += 1
		
		return res

List<List<Integer>>	List<List<Integer>> dp = new ArrayList<>();
	List<Integer> one = new ArrayList<>();
	dp.add(one);
	List<Integer> list = dp.get(i);
	list.add(j);
		dp = [] # Equivalent to List<List<Integer>>
		one = [] # Equivalent to List<Integer>
		dp.append(one) # Adding the list to dp
		list_ = dp[i] # Accessing the ith list
		list_.append(j) # Adding j to the ith list

Complex like treeSet inside Map	Map<Integer, TreeSet<Integer>> valueIndexes;	from collections import defaultdict
		from sortedcontainers import SortedSet
		
		self.value_indexes = defaultdict(SortedSet)

Integer division	T = t/10	`t = t//10`
Parse (str to int, int to str)		't = t * 10 + (int(s[0])+int(s[1])) 's = str(t)
Special functions	List Flattening -> 2D to 1D, where grid is 2D Inherit for loop while creation itself -> Find difference value is %x and store it Finding any value is not 0 -> any()	values = sorted[val for row in grid for val in row] diff = [abs(val - values[0]) % x for val in values] if any(d != 0 for d in diff):
Throw error		raise ValueError(f"Invalid input")
Pass method name and get the result as observation		Observation = tool_to_use.func(str(function_name))
If a method can return two types which can determine the next action		Agent_step: Union[AgentAction, AgentFinish] If isInstance(agent_step, AgentAction): .... If isInstance(agent_step, AgentFinish): ....

9742078725 D062 - DV

A **virtual environment** is an isolated workspace for a Python project, allowing you to install packages independently from the system-wide Python installation.

Why Use Virtual Environments?

- Avoids conflicts between package dependencies in different projects.
- Allows using different versions of the same package in different projects.
- Prevents modifying system-wide Python installations.
- Makes projects more portable and reproducible.

  Create a Virtual Environment - python3 -m venv myenv
  Activate virtural Envi - source myenv/bin/activate
  To remove envi - rm -rf myenv

Environment Variables in Python

Environment variables are **key-value pairs** used to store configuration settings outside of code. These are useful for **security-sensitive information** like API keys, database credentials, and paths.

- Create envi variable -> export SECRET_KEY=abcd1234
- Read envi variable ->import os
  secret_key = os.getenv("SECRET_KEY")

Python Function Definition Order Explained

Looking at your code where get_all_data() calls generate_frequency_data() even though it's defined later:

Why This Works

Python processes all function definitions during the module's compilation phase. All functions are registered in memory before any code actually executes. The function body isn't executed until the function is called, When Importing Functions

- When you import get_all_data from another file, the entire module is compiled first
- Both functions will be registered regardless of their order in the file

get_all_data() can safely call generate_frequency_data() even though it's defined later

Important Distinctions

- This only applies to function definitions, not variables
- Variables and standalone function calls at the module level execute in order
- Best practice is still to define functions before using them for code readability