Simulation 12

Programming Techniques · Home Practice

Assessment

Full exam: 6 exercises, 4 + 4 + 2 + 4 + 4 + 2 = 20 points

Partial I: exercises 1·3, 7 + 7 + 6 = 20 points

Partial II: exercises 4·6, 8 + 8 + 4 = 20 points

Instructions

Write independent Python solutions. Use only tools covered in the course. Read the requirements, assumptions and expected results of each exercise before beginning. Input files mentioned by name are included in the accompanying data folder. Each exercise is independent; no solution from another exercise is needed. The reference values illustrate the required behavior, but solutions must work with other valid inputs. No additional validation, formatting or functionality is required beyond what is explicitly requested.

The six exercises follow on the next pages.

Simulation 12

Full exam: 4 + 4 + 2 + 4 + 4 + 2 points. **Partial I:** 7 + 7 + 6. **Partial II:** 8 + 8 + 4.

Exercise 1 – Checking service messages (4 points – full exam)

Partial I value: 7 points

A help desk receives short status messages from an automated service. A message beginning with `OK:` reports a completed operation, but an `OK:` message containing the word `delayed` should not be considered fully successful. Messages with other prefixes are not successful either.

The desk wants a Boolean result for *each message*, identifying whether that individual message satisfies both rules. Store the results in a dictionary whose keys are the zero-based positions of the messages in the original list and whose values are `True` or `False`. The positions must refer to the original list, including unsuccessful messages; do not remove entries or count only the successful ones.

Input and assumptions

The input is a list of strings, possibly empty. Comparisons are case-sensitive. All status messages are already well-formed and need no additional validation.

Required results and operations

a) Define `check_messages(messages)`, returning one Boolean result for each input message in a dictionary indexed by the original zero-based position
b) A message qualifies only when it starts with `OK:` and does not contain the substring `delayed`
c) Print the resulting dictionary, including entries whose value is `False`; an empty input list produces an empty dictionary

Tools and restrictions Use one function, a loop, a dictionary, Boolean conditions and string operations. No numeric aggregations or external libraries are required.

Reference example

```
messages = ['OK: line 2', 'WAIT: line 5',
            'OK: delayed', 'OK: ready']

Expected output:
{0: True, 1: False, 2: False, 3: True}
```

Exercise 2 – Quality checks by laboratory (4 points – full exam)

Partial I value: 7 points

A quality-control team receives measurements from several laboratories. The data are stored in a dictionary of dictionaries: each outer key identifies a laboratory, and the associated inner dictionary maps sample codes to numerical measurements. A sample passes inspection when its measurement is at least `minimum`.

The required report contains a *separate count of passing samples for each laboratory*, including a count of zero for a laboratory with no passing samples. A second result is the total number of passing samples across all laboratories. Count each sample once, even if another laboratory uses the same sample code. Print laboratory results alphabetically and then the grand total.

Input and assumptions

Laboratory names and sample codes are strings, and measurements are numbers. A laboratory may contain no samples. There is no need to check for invalid measurements.

Count each sample exactly once; a result equal to `minimum` passes. Include empty laboratories with a count of zero and do not modify the input data.

Required results and operations

a) Define `count_passes(data, minimum)` returning a dictionary with the number of passing samples for each laboratory
b) Define `total_passes(counts)` returning the total number of passing samples across all laboratories
c) Print the individual laboratory counts in alphabetical order and then the overall total

Tools and scope Use nested loops to visit the dictionaries and their measurements, and ordinary functions, conditions and dictionaries from Lectures 1–10. Do not use third-party libraries.

Reference example

```
readings = {
    'North': {'x': 13, 'y': 8},
    'South': {'a': 16, 'b': 12, 'c': 5},
    'West': {}
}
minimum = 12

Expected output:
North: 1
South: 2
West: 0
Total: 3
```

Exercise 3 – Collect readings above a limit (2 points – full exam)

Partial I value: 6 points

Research measurements are grouped in a dictionary whose values may contain more dictionaries. At the deepest levels, the values are numbers representing individual readings. The task is to collect all readings that are *strictly greater* than a given `limit`, wherever they occur.

Write a recursive function `collect_high(data, limit)` returning a new list of the qualifying numeric values. This is a selection task: return the numbers themselves, not a count or the names of the dictionary keys. Keep their encounter order and any repeated values. For example, with limit 10, the values 12 and 15 qualify, while a value equal to 10 does not. Return an empty list if nothing qualifies.

Input and assumptions

The outer object is a dictionary. Each value is either a number or another nested dictionary. Keys are strings; empty dictionaries may occur. Return matching numeric values in dictionary insertion order, descending into nested dictionaries when encountered. Include repeated values; never return keys.

Required results and operations

a) Define `collect_high(data, limit)` and make it return the result described above for the entire structure
b) Process dictionary values at every nesting level; empty dictionaries must be handled correctly
c) Call the function(s) on the reference data and print the returned result(s) as shown

Tools and restrictions Use ordinary recursion, conditions, loops when processing nested containers, and simple string/list/dictionary operations taught in Lectures 1–10. Do not use flattening libraries or `isinstance()`; `type(x) == list` or `type(x) == dict` is sufficient.

Reference example

```
data = {'A': {'x': 5, 'y': 12}, 'B': {'z': 15, 'w': 3}}
limit = 10

print(collect_high(data, limit))

Expected output:
[12, 15]
```

Exercise 4 – Count readings above a safety threshold (4 points – full exam)

Partial II value: 8 points

A water-quality station stores integer readings in a text file. Its summary report must show how many observations were recorded, how many exceeded a fixed alert threshold of 7, and the sum of only those exceeding readings. Read `water_levels.txt` and save the report to `alerts.txt`.

Input and assumptions The supplied file contains one valid integer per nonempty line and may be empty. Exceeding means strictly greater than 7. No input validation is needed.

Required results and operations

a) Read every integer observation from `water_levels.txt`
b) Calculate the total number of readings, the number strictly above 7, and the sum of these above-threshold readings
c) Write the three results to `alerts.txt`, using the exact labels `Count`, `Alerts` and `Alert total`, one per line, and print the report

Tools and restrictions: Use ordinary text-file handling, loops and conditions. No pandas, NumPy or additional error handling.

Reference example

```
water_levels.txt:
4
10
7
11
8

alerts.txt and printed output:
Count: 5
Alerts: 3
Alert total: 29
```

Exercise 5 – Distances from a reference temperature (4 points – full exam)

Partial II value: 8 points

A laboratory has a text file of temperature readings, one intended integer per line. Some lines contain invalid text. The main program reads and validates the measurements, while an imported function calculates the absolute difference between each reading and a reference temperature of 10 degrees. The main script saves the transformed values and prints their sum.

Input and assumptions The input file exists and may contain invalid lines or blank lines. Ignore invalid or blank lines and preserve the order of valid readings.

Required results and operations

a) In `helpers.py`, define `distances(values, reference)`, returning a list of absolute differences from the reference for all supplied values
b) In `main.py`, read `temperatures.txt`, catch `ValueError` for invalid integers and call the helper once with reference 10
c) Write the resulting values to `deviations.txt`, one per line, and print their sum

Tools and restrictions: Use text-file reading and writing, loops, imports and `try-except ValueError`. No external libraries.

Reference example

```
temperatures.txt:
8
invalid
```

```
13
10

deviations.txt:
2
3
0

Expected output:
Total: 5
```

Exercise 6 – Electric scooter description (2 points – full exam)

Partial II value: 4 points

A transport company stores basic information about its vehicles. The supplied class `Vehicle` keeps a model name and offers a `.describe()` method returning a simple description of that vehicle. An electric scooter is also a vehicle, but it has one additional characteristic: its range, in kilometres.

Define `ElectricScooter` as a subclass of `Vehicle`. An electric scooter must store both the inherited model and its own `range_km`. It must also return the inherited description followed by the range in the form shown in the reference example. Do not rewrite the parent class or introduce unrelated methods; demonstrate the result by creating and printing a scooter's description.

Input and assumptions

The scooter range is a nonnegative integer and requires no validation. No exception handling or external data files are required.

Required results and operations

a) Define `ElectricScooter` as a subclass of the provided `Vehicle`; keep the parent class unchanged
b) Extend the child to store the battery range, while reusing the parent initialization appropriately
c) Override `.describe()` to extend the inherited description with the additional information in the format shown
d) Create the reference instance and print its resulting description

Tools and restrictions Use basic single inheritance, method overriding and appropriate use of `super()` to reuse initialization and description behavior. No additional libraries or exceptions are required.

Reference example

```
Provided base class:
class Vehicle:
    def __init__(self, model):
        self.model = model

    def describe(self):
        return 'Vehicle ' + self.model

Example usage:
scooter_city = ElectricScooter('City', 25)
print(scooter_city.describe())

Expected output:
Vehicle City, range 25 km
```