Solutions DIT009 H24 - Exam 3 - 2025-08-20

Q1.1: [5 pts]

i) 1 pt: Line 32

ii) 2 pt: The cause is that the variable i is na integer, and result is a String. Since Python is strongly typed, both values cannot be operated.

iii) 3 pt: To fix the error, we should convert i to string using the str() function.

Q1.2: [15t pts] +1p per correct line

```
1. A has  5  heroes:  ['Storm', '', 'Wolverine', '', 'Rogue']
2. C: result = 7 6 5 4 3 2
3. C: counter = 7
4. B: x = 7
5. B: y = 2
6. B: z = 2
7. Main: names = ['Storm', '', 'Wolverine', '', 'Rogue']
8. Main: counter = 4
9. A has  5  heroes:  ['Storm', '', 'Wolverine', '', 'Hawkeye']
10.  C: result = 8 7 6 5 4 3 2
11.  C: counter = 8
12.  B: x = 3
13.  B: y = 6
14.  B: z = 3
15.  Main:  10
```

Q2.1: [10 pts]

- 3 pt: Regex \b\d{2}:\d{2}\b
  - 21:30, 07:05
- 3 pt: \b[A-Z][ae]\w+\b
  - Neon, Keynote, Can, Send
- 4 pt: ,\s+\w+\b
  - , ran , only

Q2.2: [10 pts] +5 pts per regex and explanation.

i) `r"[^.!?]*\?"`

- `[^.!?]*` → matches any sequence of characters that are not `.`, `!`, or `?`.
- `\?` → ensures the sentence ends with a literal question mark.

ii) `r"https://[^\s]+"`

- https:// → matches the literal start of the URL.
- [^\s]+ → matches one or more non-whitespace characters (so the URL continues until a space, tab, or newline).

Q3.1: [10 pts] +5 correctness, +5 readability.

Some examples of things that will cause penalties.

- While true, poor variable names, printing instead of returning.

```
def read_animals():

    animals = []
    option = ""
    while option != "exit" :
        option = input("Enter your favourite animal: ")
        if option == "":
                print("Animal name cannot be empty")
        elif (option not in animals) and option != "exit":
                animals.append(option)

    return animals
```

Q3.2: [10 pts] +5 correctness, +5 readability.

Some examples of things that will cause penalties.

- Incorrect use of nested loops, poor variable names, printing instead of returning, wrong use of exception.

```
def describe_animals(adjectives, animals):

   if (not adjectives) or (not animals):
       raise ValueError("Lists cannot be empty.")

   else:
       described_animals = []
       for adjective in adjectives:
           for animal in animals:
               described_animals.append(adjective + " " + animal)

       return described_animals
```

Q3.3: [10 pts] +6 explanation, +4 printed.

i) The function receives a list of animals, and then returns a dictionary where the names of the animals are categorised based on how long the name is. They are categorised into short (less than 5 characters), medium (5-7 characters), long (greater than 7 characters).

ii) Printed:

- {'short': ['cat', 'dog', 'cow'], 'medium': ['tiger'], 'long': ['elephant', 'alligator']}
- {'short': ['bat', 'ox', 'emu'], 'medium': ["whale"], 'long': ['hippopotamus']}
- {}

Q3.4: [10 pts] +2-3 pts per code smell identified.

Code smell 1:

- **i) Poor / non-descriptive naming:** Names don't convey intent, lowering readability and making maintenance harder.
- **ii) Lines:** 1–4 (`doStuff`, `x`, `d`, `i`, `l`)
- iii) We fix it by Rename to meaningful identifiers, e.g. `categorize_by_length(names)`, `buckets`, `name`, `length`.

Code Smell 2:

- **i) Duplicated code (key creation + append per bucket):** the pattern "ensure key exists → append". Code is not DRY.
- **ii) Lines:** 6–8, 10–12, 14–16
- **iii) Refactor by** pre-initializing buckets: `{"short": [], "medium": [], "long": []}` and just append, **and** eliminating the "if key not in d" checks.

Code Smell 3:

- **i) Magic numbers** (`5`, `7`):Bare numeric thresholds obscure meaning and are hard to change consistently.
- ii) **Lines:** 5, 9, 13
- iii) **Refactor by** replacing with named constants or parameters, e.g. `SHORT_LT = 5`, `MEDIUM_LT = 8` (so "medium" is `< 8`), or make them function parameters with defaults.

Code Smell 4:

- **i) Redundant conditions / overly specific bounds:** After `if l < 5`, the `>= 5` part is redundant; the last `elif l > 7` can be a simple `else` if the previous branch handles `l < 8`.
- **ii) Lines:** 9 (`elif l >= 5 and l <= 7:`), 13 (`elif l > 7:`)
- **iii) Refactor by using** a clean, mutually exclusive chain, e.g.:

    - `if length < SHORT_LT: …`
    - `elif length < MEDIUM_LT: …`
    - `else: …`

Code Smell 5:

- **Missing documentation / type hints** (style & clarity). No docstring or type hints makes intent and expected types unclear.

- **Lines:** 1 (function signature)
- **Refactor by adding** a docstring to functions and blocks of code.

Q4.1: [10 pts] +2-3 pts per code smell identified.

Note: It must use recursion, otherwise the question received zero points.

Readability: 4 pts
- Organisation of the code, naming of variables and function, following conventions, easy to understand code.

Correctness: 6 pts
- [1 pts] Works for the base case of not found
- [1 pts] Works for the base case of only one element
- [4 pts] Recursive step is done correctly

Examples of penalties:
- [-2 pts] Any input or printing inside function.
- [-2 pts] Incorrect function parameters. OK if index=0.
- [-2 pts] Uses Exceptions (they are unnecessary).
- [-3 pts] Missing returns in the function

```
 1 def calc_even_avg(list, sum, count, index):
 2   if index == len(list):
 3       avg = 0
 4       if count > 0:
 5           avg = sum / count
 6       return avg
 7
 8   else:
 9       current = list[index]
10       if current % 2 == 0 and current > 0:
11           return calc_even_avg(list, sum + current, count + 1, index + 1)
12       else:
13           return calc_even_avg(list, sum, count, index + 1)

print(calc_even_avg([4, -3, 7, 10, 2], 0, 0, 0))      # 5.333
print(calc_even_avg([-2, -4, 3, 5], 0, 0, 0))         # 0
print(calc_even_avg([8, 12, 15, 20], 0, 0, 0))        # 13.333
print(calc_even_avg([10, 2, 0, -15, -2], 0, 0, 0))    # 6
print(calc_even_avg([], 0, 0, 0))                     # 0
```

Q4.2: [10 pts] Graded based on correct sequence (+3) and variable values (+7)