Lab 1 Exam Questions

December 2020

1 Assembly Emulator

In this task, you are expected to write a function (`decode_and_execute(Memory, Instruction)`) that will execute a hypothetical assembly language. `decode_and_execute` is expected to decode the given instruction and execute it. The parameters of function are:

- `Memory`, which is a list of numbers (int or float) with size 10. Initially, the `Memory` is assumed to be `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`. `Memory` is accessible by addresses which are the indexes such that `Memory[address]` is the content of the `Memory` at address `address`.
- `Instruction`, which is a string containing an instruction.

Here are the instructions that the function is expected to handle:

Instruction	Description
`"CLRMEM"`	Set every element in the `Memory` to 0.
`"SUB address1, address2"`	Value at `address2` (i.e. `Memory[address2]`) is subtracted from the value at `address1` (i.e. `Memory[address1]`) and the result is stored at `address1` (i.e. `Memory[address1]`)
`"LOAD address, number"`	Loads the given `number` value into the location at the address (i.e. `Memory[address]`)
`"SUBI address, number"`	`number` is subtracted from the value at `address` and the result is stored at `address` (i.e. `Memory[address]`)
`"DIVI address, number"`	Value at `address` is divided by `number` using floating division (even if the values are integer) and the result is stored at `address` (i.e. `Memory[address]`)

The function return a list of error messages. The list contains one or more of the following error messages (the function should not return `Memory`):

- `"NO ERROR"` if there are no errors.

- `"DIVISION BY ZERO"` if the divisor is zero.
- `"UNKNOWN INSTRUCTION"` if the instruction is not known.
- `"INVALID ADDRESS"` if an invalid address is provided. I.e. either the address is not a number or it is not in the range [0, 9].
- `"NOT A NUMBER"` if an instruction expecting an integer is provided not a number.

If there are no errors, the function returns `["NO ERROR"]`. When an invalid instruction is fed to the function, (no matter what the other errors might be), `["UNKNOWN INSTRUCTION"]` is returned. For the remaining error cases, the function should output a list of the errors. For instance, `"DIV 12, 0"` leads to both `"INVALID ADDRESS"` and `"DIVISION BY ZERO"` errors and the function should return `["INVALID ADDRESS", "DIVISION BY ZERO"]` (**The order of the errors in the list is not important**). Similarly `"SUB 45, 2.0"` returns `["INVALID ADDRESS", "NOT A NUMBER"]`. If an instruction leads to an error multiple times, it needs to be included in the list multiple times. For instance, `"SUB 12, 45"` instruction should return `["INVALID ADDRESS", "INVALID ADDRESS"]`."

In case of an error, the corresponding instruction is not executed and the memory is left intact.

Let us look at some sample runs. Initially, `Memory` is `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`. And the following instructions have been run sequentially. Function outputs and the `Memory` content after the function execution are as follows:

- `decode_and_execute(Memory, "LOAD 1, 456")` → returns `["NO ERROR"]` and `Memory` content: `[0, 456, 0, 0, 0, 0, 0, 0, 0, 0]`
- `decode_and_execute(Memory, "LOAD 1, 25.0")` → returns `["NOT A NUMBER"]` and `Memory` content: `[0, 456, 0, 0, 0, 0, 0, 0, 0, 0]`
- `decode_and_execute(Memory, "LOAD 45, ceng111")` → outputs `["INVALID ADDRESS", "NOT A NUMBER"]` and `Memory` content: `[0, 456, 0, 0, 0, 0, 0, 0, 0, 0]`
- `decode_and_execute(Memory, "LOAD 9, 100")` → returns `["NO ERROR"]` and `Memory` content: `[0, 456, 0, 0, 0, 0, 0, 0, 0, 100]`
- `decode_and_execute(Memory, "SUB 1, 9")` → returns `["NO ERROR"]` and `Memory` content: `[0, 356, 0, 0, 0, 0, 0, 0, 0, 100]`
- `decode_and_execute(Memory, "SUB 10, 9")` → returns `["INVALID ADDRESS"]` and `Memory` content: `[0, 356, 0, 0, 0, 0, 0, 0, 0, 100]`
- `decode_and_execute(Memory, "DIVI 9, 25")` → returns `["NO ERROR"]` and Memory content: `[0, 356, 0, 0, 0, 0, 0, 0, 0, 4.0]`

- `decode_and_execute(Memory, "DIVI 100, 0")` → returns `["INVALID ADDRESS", "DIVISION BY ZERO"]` and Memory content: `[0, 356, 0, 0, 0, 0, 0, 0, 0, 4.0]`
- `decode_and_execute(Memory, "SUBI 0, 540")` → returns `["NO ERROR"]` and Memory content: `[-540, 356, 0, 0, 0, 0, 0, 0, 0, 4.0]`
- `decode_and_execute(Memory, "MUL 152, 540")` → returns `["UNKNOWN INSTRUCTION"]` and Memory content: `[-540, 356, 0, 0, 0, 0, 0, 0, 0, 4.0]`
- `decode_and_execute(Memory, "CLRMEM")` → returns `["NO ERROR"]` and Memory content: `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`
- `decode_and_execute(Memory, "SUB 45, 1.0")` → returns `["INVALID ADDRESS", "INVALID ADDRESS"]` and Memory content: `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`

Specifications:

- You must not import any modules.
- PLEASE REMOVE **print** functions, if any, while passing to next question and evaluating.
- The usage of **input()** **map()**, **reduce()**, **filter()** functions is strictly forbidden.

  The usage of iteration (**for**, **while** and **list/set comprehension**) is strictly forbidden.
- If need be, you can use **dir** and **help** functions to look at the available functions and their descriptions for each data type.

```
def decode_and_execute(MEMORY, command):
    available_commands = ["CLRMEM", "LOAD", "SUBI", "DIVI", "SUB"]
    error_list=[]
    if command == "CLRMEM":
        pass
    else:
        parts = command.split(",")
        parts2 = parts[0].split(" ")
        command = parts2[0]
        part1 = parts2[1]
        part2 = parts[1][1:]
        if not command in available_commands:
            return ["UNKNOWN INSTRUCTION"]
        if part1.isdigit():
            if 0<=int(part1)<=9:
                pass
            else:
                error_list.append("INVALID ADDRESS")
        else:
            error_list.append("INVALID ADDRESS")
```

```
        if "-" in part2:
            temp_part2 = part2[1:]
        else:
            temp_part2 = part2

        if temp_part2.isdigit():
            if command == "SUB":
                if 0<= int(part2) <=9:
                    pass
                else:
                    error_list.append("INVALID ADDRESS")
            elif command == "DIVI":
                if -0.00000001< float(part2) < 0.000000001:
                    error_list.append("DIVISION BY ZERO")
        else:
            if command == "SUB":
                error_list.append("INVALID ADDRESS")
            else:
                error_list.append("NOT A NUMBER")

    if command == "CLRMEM":
        MEMORY[0]=0;MEMORY[1]=0;MEMORY[2]=0;MEMORY[3]=0;
        MEMORY[4]=0;MEMORY[5]=0;MEMORY[6]=0;MEMORY[7]=0;
        MEMORY[8]=0;MEMORY[9]=0
    elif command == "LOAD" and len(error_list) == 0:
        MEMORY[int(part1)] = int(part2)
    elif command == "SUB" and len(error_list) == 0:
        MEMORY[int(part1)] -= MEMORY[int(part2)]
    elif command == "SUBI" and len(error_list) == 0:
        MEMORY[int(part1)] -= int(part2)
    elif command == "DIVI" and len(error_list) == 0:
        MEMORY[int(part1)] /= int(part2)


    if len(error_list) == 0:
        return ["NO ERROR"]
    else:
        return error_list
```

2 Arithmetic without Numerical Types

Write a function for string addition (for base-10), namely `string_op(str1, str2)`, where the parameters `str1` and `str2` are strings that contain **positive numbers** which have at most 2-digits. The function returns the result in string form.

For instance, the function call `string_op("29","46")` should implement the following and return `"75"`:

```
    "29"
+   "46"
-------
    "75"
```

What `string_op` function should do is to first look at the least significant digits and perform addition on them and then continue to the other digits from right to left. While doing so, you need to take the carry into account: In the example above, `"9"+"6"` results in `"15"`, the function should write `"5"` to the result and transfer `"1"` as the carry for the next digit calculation. In the next digits, `"2"+"4"+"1"` is performed, resulting in `"7"`. Since all digits are consumed, the function should complete summation and return `"75"` for our example.

In order to simplify the string-wise addition, a function called `string_sum(x,y)` has been defined in `CENG111` package (and imported at the top of your template code). `string_sum(x,y)` takes two **single-digit** strings and returns the result of the summation with carry information. For instance, you can directly call the function as `string_sum("5","9")` and it returns the list `["1", "4]` where the first is the carry information and the second is the summation result. Similarly, `string_sum("5","3")` returns `["0", "8"]` .

Specifications:

- You are expected to implement `string_op(str1,str2)` function with string operations, if-else structures, function definitions etc.
- Usage of the operators `+`, `-`, `/`, `//`, `%`, `**` (with strings, integers, floats, lists, tuples) is strictly forbidden. If any of these are detected in your code, you will get a grade of 0.
- Similarly, the functions: `__add__`, `__radd__`, `__sub__`, `__rsub__`, `__mod__`, `__rmod__` `__str__`, `__int__`, `__divmod__`, `__neg__`, `__div__`, `__rdiv__`, `__mul__`, `__rmul__` etc with lists, strings, integers, floats, tuples are strictly forbidden. If any of these are detected in your code, you will get a grade of 0.
- The usage of **negative indexes** for lists, strings, tuples is strictly forbidden (due to `-`).

- The usage of `int()`, `str()`, `map()`, `reduce`, `filter` functions is strictly forbidden. If any of these are detected in your code, you will get a grade of 0.
- The usage of iteration (`for`, `while` and list/set comprehension) is strictly forbidden.
- You must not import any modules except CENG111 (already imported).
- PLEASE REMOVE print functions, if any, while passing to the next question and evaluating.
- If need be, you can use **dir** and **help** functions to look at the available functions and their descriptions for each data type.
- Since your are not allowed to use `+` operator for string concatenation, we would like introduce some other ways to carry out this operation. First you may use `join()` function of the string type. `"".join("Ceng", "111")` returns `"Ceng111"`. As second method, you may leverage `format` function of the string type. `"{0}{1}".format("Ceng","111")` returns `"Ceng111"`.
- You may split a string into sub-strings via `split` function of the string data type. For instances, `"Ceng 111".split(" ")` yields the following list `["Ceng", "111"]`.

Sample inputs and outputs are as follows:

- `number_operation("45","33")` → `"78"`
- `number_operation("75","44")` → `"119"`
- `number_operation("19","5")` → `"24"`
- `number_operation("8","1")` → `"9"`
- `number_operation("0","0")` → `"0"`

```
from CENG111 import string_sum
# DO NOT CHANGE THE ABOVE LINE, YOU CAN DIRECTLY CALL THE GIVEN
    FUNCTION AS:
# string_sum("5","9") and it will return the list of strings
    ["1","4"] for this example
# WRITE YOUR CODE BELOW.

def string_op(number1, number2):
    result=["0", "0", "0"]
    if len(number1) == 2 and len(number2) == 1:
        carry, value = string_sum(number1[1], number2[0])
        result[2] = value
        carry, value = string_sum(number1[0],carry)
        result[1] = value
        result[0] = carry

    elif len(number1) == 2 and len(number2) == 2:
        carry, value = string_sum(number1[1], number2[1])
        result[2] = value

        old_carry, old_value = string_sum(number1[0], carry)
        carry, value = string_sum(number2[0], old_value)
        result[1] = value
        if old_carry == "1" or carry == "1":
            carry = "1"
        carry, value = string_sum(result[0], carry)
        result[0] = value
    elif len(number1) == 1 and len(number2) == 2:
        carry, value = string_sum(number1[0], number2[1])
        result[2] = value
        carry, value = string_sum(number2[0], carry)
        result[1] = value
        result[0] = carry
        pass
    elif len(number1) == 1 and len(number2) == 1:
        carry, value = string_sum(number1[0], number2[0])
        result[2] = value
        result[1] = carry

    if result[0] == "0": result.pop(0)
    if result[0] == "0": result.pop(0)

    return "".join(result)
```

```
# Content of CENG111.py

def string_sum(x,y):
    if len(x) <= 0 or len(x) >1:
        return ["", ""]
    if len(y) <= 0 or len(y) >1:
        return ["", ""]
    res=ord(x)-48+ord(y)-48
    return chr(res//10+48), chr((res%10) +48)
```