COMP10001 Foundations of Computing
Semester 2, 2022

Tutorial Solutions: Week 8

— VERSION: 1480, DATE: SEPTEMBER 12, 2022 —

Discussion

1. What is a "library"? How do we access them?

   A: *A library contains a group of methods or variables which can extend Python to perform more diverse operations. To access the contents of a library, we import it using the* `import` *keyword. This adds the module to our program's namespace so it can be used.* `import <library>` *imports the entire library, with any methods or constants it contains.* `from <library> import <name>` *imports a particular method or value from that library.*

2. What is a "`defaultdict`"? How do we initialise and use it?

   A: *A* `defaultdict` *is a data type contained in the* `collections` *library which has the same behaviour as a dictionary with the added functionality of initialising new keys to a default value when trying to use them. To use a defaultdict, you must first import it by including the line* `from collections import defaultdict` *at the top of your program. Then, to create a defaultdict, you call* `defaultdict(<type>)` *where* `type` *is the data type of the values in the dictionary (the keys can be of any immutable type). When requesting or updating values which haven't been previously set, rather than raising a* `KeyError`*, Python will initialise the value to the "zero value" of that data type (zero for numbers, an empty container for strings, lists, dictionaries ect.)*
   Defaultdicts are especially useful for counting, as you can increment for each time you see something without worrying about initialising it if it's the first time you've seen that value.

   Now try Exercise 1

3. What is a "bug"? What are some debugging strategies which we can use when we find an error?

   A: *In computing, a bug is an error in code which causes a program to not run as intended. After finding that there is a bug in our program by running test cases (discussed later), strategies to fix bugs include tracing code to find sections relevant to the error encountered and using diagnostic print statements in these sections to check the value of variables during execution. Where values are unexpected or code is not running in the way it is supposed to, we've found where the error is and can write new code to fix the problem.*

4. What are the three types of errors we've learned about?

   A: *The three errors are (1) Syntax error: where the code won't run due to incorrect use of symbols in the programming language; (2) run-time error: where the code runs but a problem is encountered during execution causing the program to crash; and (3) logic error: where the code runs to completion without problem, but the result is not what the developer intended.*

5. How can we use testing to find bugs or confirm our code runs properly? What are some strategies we can adopt to write comprehensive test cases for our code?

   A: *To ensure that code is correct, it is important to run it under a range of scenarios to ensure that it not only works properly in a specific case, but in* ***all*** *cases. To write test cases, we should think about different possible inputs our code could receive and write test cases to cover as many of them as possible. This does not mean writing a test case for every possible input, rather a test case for each* ***category*** *of input, especially testing any "corner cases" which are at the limits of the code's specification.*

   Now try Exercises 2 – 4

Exercises

1. Rewrite the following with a default dictionary

```
my_dict = {}
for i in range(10):
    if i % 3 in my_dict:
        my_dict[i % 3].append(i)
    else:
        my_dict[i % 3] = [i]
```

A:

```
from collections import defaultdict

my_dict = defaultdict(list)
for i in range(10):
    my_dict[i % 3].append(i)
```

2. Find the errors in the following programs, classifying them as (a) syntax, (b) runtime or (c) logic errors. Fix them with a correct line of code.

(a)
```
def disemvowel(text):
    """ Returns string `text` with all vowels removed """
    vowels = ('a', 'e', 'i', 'o', 'u')
    answer = text[0]
    for char in text:
        if char.lower() is not in vowels:
            answer = char + answer
    print(answer)
```

A: *Below is the line number of the error, the type of error and a replacement line of code to fix it.*

- *line 4; logic/run-time (if empty string);* `answer = ''`
- *line 6; syntax;* `if char.lower() not in vowels:`
- *line 7; logic;* `answer = answer + char` *or* `answer += char`
- *line 8; logic;* `return answer`

(b)
```
def big-ratio(nums, n):
    """ Calculates and returns the ratio of numbers
    in list `nums` which are larger than `n` """
    n = 0
    greater_n = 0
    for number in nums:
        if number > n:
            greater_n += 1
            total += 1
    return greater_n / total

nums = [4, 5, 6]
    low = 4
print(f"{100*big_ratio(nums, low)}% of numbers are greater than {low}")
```

A:

- *line 1; syntax;* `def big_ratio(nums,n):`
- *line 4; logic/(run-time as well since it would cause error as* `total` *is undefinedl);* `total = 0`
- *line 9; logic; remove one level of indentation (outside* `if` *block)*
- *line 13; syntax; remove indentation*

3. The function below is supposed to take a list of integers and remove the negative integers from the list, however, it is not working as intended.

- Write down three test cases that could be useful for function verification or finding bugs.
- Debug the associated code snippet to solve the problem.

Write a function that takes a list of integers and removes the negative integers from the list.

```
def remove_negative(nums):
    for num in nums:
        if num < 0:
            nums.remove(num)
```

A: *Test cases to consider include: The empty list `[]`, a list with no negative numbers `[0,1,2]` and a list with only negative numbers `[-1, -2, -3]`.*
The debugging process will be different for everyone, but here is an example:
Begin by observing the function's failure to process the following test case.

```
lst = [-1, -2, 3]
remove_negative(lst)
print(lst)
```

Include a print statement to observe the values of the variables within the for loop.

```
def remove_negative(nums):
    for num in nums:
        print(num, nums)
        if num < 0:
            nums.remove(num)
```

Repeating the above test case, one will find that `num` takes the value -1 and 3, but skips -2 entirely. With any luck, this will lead to the recollection/realisation that it is dangerous to remove elements from list whilst iterating over them, as Python can skip elements. Instead, the following solution may be attained:

```
def remove_negative(nums):
    to_remove = []
    for num in nums:
        if num < 0:
            to_remove.append(num)

    for num in to_remove:
        nums.remove(num)
```

4. This question is based on a previous exam question. The code below is intended to validate a data entry, being a list of the following string elements.

   (a) a *staff ID*, valid if it is a 5 digit number (e.g. `"00520"` or `"19471"`)

   (b) a *first name*, valid if non-empty and only containing alphabetical letters

   (c) a *password*, valid if including at least one lower-case letter, one upper-case letter, and one punctuation mark from the following `[',', '.', '!', '?']`

   The function should return `True` if the data entry contains entirely valid values (according to the above rules) and `False` if any of the fields are invalid. A valid data example is: `['10001', 'Chris', 'Comp!']`

```
STAFFID_LEN = 5

def validate(data):
    staffid = data.pop(0)
    if not 10**(STAFFID_LEN-1) <= int(staffid) < 10**STAFFID_LEN:
        return False

    first_name = data.pop(0)
    if not first_name and first_name.isalpha():
        return False

    password = data.pop(0)
    contains_lower = contains_upper = contains_punct = False
    for letter in password:
        if letter.islower():
            contains_lower = True
        elif letter.isupper():
            contains_upper = True
        elif not letter.strip(',.!?␣'):
            contains_punct = True

    if not contains_lower and contains_upper and contains_punct:
        return False

    return True
```

The provided code is imperfect, in that sometimes it validates and invalidates correctly, but sometimes it misclassifies valid data as invalid and invalid data as valid. Your task is to:

(a) Provide an example of valid data that is correctly classified as such by the provided code (i.e. valid data input where the return value is `True`).

(b) Provide an example of invalid data that is correctly classified as such by the provided code (i.e. invalid data input where the return value is `False`).

(c) Provide an example of invalid data that is *in*correctly classified as a valid by the provided code (i.e. valid data input where the return value is erroneously `True`).

(d) Provide an example of valid data that is *in*correctly classified as an invalid by the provided code (i.e. invalid data input where the return value is erroneously `False`).

A: *(a) Valid data that is correctly classified:* `['12345', 'Kim', 'Ron!']` *(Function returns* `True` *correctly - input is valid)*

(b) Invalid data that is correctly classified: `['12345', 'Kim', 'RON!']` *(Function returns* `False` *correctly - password has no lowercase)*

*(c) Invalid data that is *in*correctly classified:* `['12345', '', 'Ron!']` *(Function returns* `True` *incorrectly since empty name is invalid but check is incorrect)*

*(d) Valid data that is *in*correctly classified:* `['00117', 'Kim', 'Ron!']` *(Function returns* `False` *incorrectly since staff ID is correct but check is not)*

Problems

1. Write a function which takes a string as input and returns a sorted list of the words which occur only once in the string. Try using a defaultdict and list comprehensions in your solution.
`once_words('the cat in the hat is a cat')` should return `['a', 'hat', 'in', 'is']`.

A:

```
from collections import defaultdict

def once_words(text):
    freqs = defaultdict(int)
    for word in text.split():
        freqs[word] += 1
    return sorted([word for word in freqs if freqs[word] == 1])
```

2. Write a function which takes a string containing an FM radio frequency and returns whether it is a valid frequency. A valid frequency is within the range 88.0-108.0 inclusive with 0.1 increments, meaning it must have only one decimal place. `valid_fm('103.14')` should return `False`.

A:

```
def valid_fm(freq):
    """ Takes an FM frequency as a string and returns a boolean
    value indicating whether it's a valid frequency or not. """

    # Checks length of string
    if len(freq) != 4 and len(freq) != 5:
        return False

    # Checks characters conform to a number
    if not freq[:-2].isdigit() or freq[-2] != "." or not freq[-1].isdigit():
        return False

    # Returns based on final range check
    return 88.0 <= float(freq) <= 108.0
```