Informatics I – HS18

Exercise 6

Submission Details

- **Submission Format:** ZIP file containing your .py solution files
- **Submission Deadline:** 12:00, Tuesday 6th November
- The name of the ZIP file must have the following format:
  firstname_lastname_studentidentificationnumber_info1_exercise_6.zip,
  e.g. *hans_muster_12345678_info1_exercise_6.zip*.
- Your ZIP file should contain the following files: `task_1_test.py`,`task_2_test.py`,`task_3_test.py` and `task_4_test.py`. **Do not rename these files.**
- **Important:** Please follow the naming conventions very strictly, otherwise we will not be able to grade your submission.
- Please submit the assignment using OLAT. You may upload multiple solutions, but note that **only the last submission** will be evaluated.
- Your submissions will be tested on systems running **Python 3.7.X**. Make sure that your solutions work on this version.

Introduction

In this exercise, the tables have turned – we write the code and you test it! In the following pages, we will present some code that needs to be tested. Each task will describe the code we wrote, its expected inputs and outputs and any additional information you may need to write unit tests.

How many tests are enough?

For the exercise, we do not ask you to write N tests just for the sake of writing tests. The goal is to write enough tests such that **you** are confident that any implementation that passes your tests is actually a correct one. Instead of thinking about the number of tests, think about the different scenarios that you need to cover with them, think about boundaries of natural domains, think about the problem domain, try to cover as many of the branches in the program as possible.

To illustrate this, have a look at the following code snippet. You can see a bad example of tests where only one class of inputs is tested (all tested input values are inside the bounds), whereas in the good example different classes of inputs are tested (outside the bounds, lower & upper bound, inside the bounds).

```python
def is_in_range_0to100(a):
    return 0 <= a <= 100


# Bad tests (only one of these is necessary): these count as a single
    class of inputs
is_in_range_0to100(20)
is_in_range_0to100(45)
is_in_range_0to100(90)

# Better tests: these count as different classes of inputs (lower bound,
    inside the bounds, upper bound, outside the bounds)
is_in_range_0to100(-1)
is_in_range_0to100(0)
is_in_range_0to100(50)
is_in_range_0to100(100)
is_in_range_0to100(101)
```

Listing 1: Example of bad and good unit test cases.

Testing tools

We provided you with the boilerplate code necessary to write tests using the `unittes` framework. However, It's up to you if you prefer using a testing framework (*Defining Unit Tests in* `unittest` from the slides) or using simple assert statements (e.g. *Poor Man's Testing "Framework"*). There is no *bonus* or *malus* to the evaluation either way.

1 Task: Quicksort (2 Points)

The following code snippet shows a sorting algorithm called Quicksort (https://en.wikipedia.org/wiki/Quicksort) that takes a list of numbers as input and returns the same list, sorted in ascending order.

Note: You do not have to test the following cases:

- Lists with types other than numbers (e.g. strings)
- Lists with mixed types (e.g. strings and numbers)
- Anything that is not a list, except None

Task: Write unit tests that test the function `quick_sort`.

```python
def quick_sort(array):
    """
    Takes as input a list of numbers, sorts it in ascending order and
    returns the sorted list.

    :param array: A list of numbers
    :return: The same list, sorted in ascending order
    """

    if not array:
        return array

    less = []
    equal = []
    greater = []

    pivot = array[0]
    for x in array:
        if x < pivot:
            less.append(x)
        if x == pivot:
            equal.append(x)
        if x > pivot:
            greater.append(x)

    return quick_sort(less) + equal + quick_sort(greater)
```

Listing 2: Quicksort algorithm.

2 Task: List Duplicates (1.5 Points)

The following function takes as input a list of numbers and returns a list of tuples, where each tuple contains, the value of the duplicate and the indexes at which the value occurs.

Note: You do not have to test the following cases:

- Lists with types other than numbers (e.g. strings)
- Lists with mixed types (e.g. strings and numbers)
- Anything that is not a list, except None

Task: Write unit tests that test the function `list_duplicates`.

```python
def list_duplicates(sequence):
    """
    Takes as input a list of numbers and returns a list of tuples,
    where each tuple contains, the value of the duplicate and the indexes
    at which the value occurs.

    E.g. list_duplicates([1, 2, 1, 2, 3, 4, 5, 6, 7, 4]) == [(1, (0, 2)),
    (2, (1, 3)), (4, (5, 9))]

    :param sequence: A list of numbers
    :return: List of tuples of duplicates and their indexes.
    """
    if not sequence:
        return sequence

    occurrences = {}
    for idx, element in enumerate(sequence):
        if element in occurrences:
            occurrences[element].append(idx)
        else:
            occurrences[element] = [idx]

    return [(element, tuple(occurrences[element])) for element in
    occurrences if len(occurrences[element]) > 1]
```

Listing 3: List duplicates.

3 Task: Accounts (3 Points)

The following code snippet lists two functions. `transfer_from_to` takes as input a dictionary containing account names and their balance, as well as the sender's and receiver's account name and the amount that should be transferred. It returns a new updated dictionary with the new account balances. The money is only transferred if both accounts exist and there is enough money in the sender's account. It is only possible to transfer a non-negative amount of money. If for any of the mentioned reasons, the transfer fails, the function just returns the unmodified accounts dictionary (i.e. a copy thereof). `create_account` also takes as input a dictionary with account names and balances, as well as the name and initial balance of the new account that should be created. It returns a new updated dictionary that includes the newly created account, if the creation was successful. The creation is only successful if no account with the given name exists yet, the given name is not the empty string and the initial balance is non-negative. Otherwise, the function just returns the unmodified accounts dictionary (i.e. a copy thereof).

Note: You do not have to test the following cases:

- Lists with the wrong types (e.g. a string where a number is expected)
- `accs` is None

Task: Write unit tests that test the functions `transfer_from_to` and `create_account`.

```python
def transfer_from_to(accs, account_from, account_to, value):
    """
    Transfers money from one account to another, given that both accounts
    exist and there is enough money in the sender's account.

    :param accs: A dictionary containing all accounts and their balance
    :param account_from: A string specifying the sender's account
    :param account_to: A string specifying the receiver's account
    :param value: The amount that should be transferred
    :return: A new, updated accounts dictionary (The account dictionary
    should not be modified directly, instead a copy should be created,
    modified and returned.
    """

    accounts = dict(accs)

    if account_from in accounts and account_to in accounts:
        if 0 < value <= accounts[account_from]:
            accounts[account_from] -= value
            accounts[account_to] += value

    return accounts


def create_account(accs, name, initial_balance):
    """
    Creates a new account with the given name and initial balance. If an
    account with the given name already exists, the new account is not
    created.

    :param accs: A dictionary containing all accounts and their balance
    :param name: The name of the new account to create (must be a non-empty string)
    :param initial_balance: The initial balance of the account (must be non-negative)
    :return: A new, updated accounts dictionary (The account dictionary should not be modified directly, instead a copy should be created, modified and returned.
    """
    accounts = dict(accs)

    if name and name not in accounts and initial_balance >= 0:
        accounts[name] = initial_balance

    return accounts
```

Listing 4: Accounts.

4 Task: Swap Case - Blackbox testing (1.75 Points)

While in the previous examples, you've had actual code to test, this time around you will only receive the signature and documentation of a function.

Task: Write unit tests that test the functions `swap_case`.

```python
def swap_case(s):
    """
    Swaps the case of a String changing upper case to lower case,
    and lower case to upper case.
          * swap_case("The dog has a BONE") = "tHE DOG HAS A bone"

    :param s: the string to swap case, may be None
    :return: the changed String, None if None input
    """
    pass
```

Listing 5: `swap_case` signature and docstring.