Semester 2, 2019: Lab 10

S2 2019

Lab 10

This lab contains some examples of the type of problems you may encounter on the final exam. The lab is divided into two parts: The first part contains questions on reading and understanding python code, the second part contains programming problems. Problems are presented in no particular order. Don't assume that the first is easier than the second, and so on. There is likely to be more problems in this lab than you can finish in one 2-hour lab; if you don't finish all of them during the lab, continue practicing on the remaining problems outside of lab time.

To check your answers to the exercises in the first part, compare with those of other students in the lab, and ask your tutor.

Exercise 1

Each of the following pieces of python code attempt to compute and print the maximum difference between any pair of numbers in a list `x`. (Note that the maximum difference is always greater than or equal to zero, which is the difference between any number and itself.) For each one, you should determine whether it is correct, meaning that it runs without error and prints the correct value. If it is not, explain precisely what is wrong with it. Assume that variable `x` is defined in the global namespace and that its value is a non-empty list of numbers.

(a)

```
def find_max(x, y):
    c = []
    for i in x:
        for j in y:
            c.append(x[i] - y[j])
    return max(c)

print("The max difference is ", find_max(x, x))
```

(b)

```
def find_max(x, y):
    d = 0
    for i in x and j in y:
        d = max(d, i - j)
    return d

print("The max difference is ", find_max(x, x))
```

(c)

```
def find_max(y):
    z = x[0] * y
    for i in range(len(x)):
        x[i] = y * x[i]
        if x[i] > z:
            z = x[i]
    return z

print("The max value is ", find_max(1))
print("The min value is ", find_max(-1))
print("The max difference is ", find_max(1) - find_max(-1))
```

(d)

```
def find max(x):
    x = x.sort()
    return x[-1] - x[1]

print("The max difference is ", find max(x))
```

Exercise 2

We call a dictionary invertible if every key in it maps to a unique value, or in other words, for every value appearing in the dictionary, there is only one key that maps to that value. Below are three attempts to define a function `is_invertible(adict)`, which takes as argument a dictionary and returns `True` if the dictionary is invertible, and `False` otherwise.

For example, `is_invertible({ 'a' : 'b', 'b' : 'e', 'c' : 'f' })` should return `True`, but `is_invertible({ 'a' : 'b', 'b' : 'e', 'c' : 'b' })` should return `False`, because keys `'a'` and `'c'` both map to the same value, `'b'`.

For each of the functions below, determine whether it is correct or not. Correct mean that the function runs without error and returns the right answer for any dictionary. If a function is not correct, explain precisely what is wrong with it.

(a)

```
def is_invertible(adict):
    return sorted(adict.keys()) == sorted(adict.values())
```

(b)

```
def is_invertible(adict):
    d = {}
    for value in adict.values():
        if value in d:
            return False
        else:
            d[value] = 1
    return True
```

(c)

Note: This implementation of the function uses another function, `make_inv_dict`, which is also defined below.

```
def is_invertible(adict):
    inv_dict = make_inv_dict(adict)
    return adict == inv_dict

def make_inv_dict(adict):
    if len(adict) > 0:
        key, val = adict.popitem()
        adict = make_inv_dict(adict)
        if val not in adict.values():
            adict[key] = val
        return adict
    else:
        return {}
```

Exercise 3

Here is a function that takes as argument a sequence:

```
def funX(a_list):
    index = 0
    while index < len(sorted(a_list)) // 2:
        index = index + 1
    return sorted(a_list)[index]
```

(a) Explain what `funX` does *in general.* A good answer is one that describes the purpose of the function - something you would write in its docstring.

(b) `funX` is unnecessarily inefficient. Rewrite the function so that it does the same thing, but as efficiently as possible.

Exercise 4

Here is another function that takes a sequence as argument:

```
def funY(x):
    i = 0
    while i < len(x):
        i = i + x[i] % len(x)
    return i
```

(a) Give an example, if possible, of an argument sequence that causes `funY` to get stuck in an infinite loop, as well as an argument sequence for which the function executes without error and returns a value.

(b) What are the runtime errors that can occur in `funY`, assuming the argument is of type `list`? For each error, give an example of an argument list that causes the error.

Exercise 5

Here is a function and its doc string:

```
def smallest_non_negative(number_list):
    '''Argument is a list of numeric values (integer or float).
    Returns the smallest non-negative value in the list,
    and zero if there is no such value.'''
```

Write at most four test cases for this function. For each test case, you must write down the input (argument to the function) and the expected return value. Your test cases should cover all relevant corner cases.

Programming problems

For each problem there is a problem description and a skeleton file for you to write your solution into. This file also has a testing function that runs several test cases on your functions (like you have seen in the homeworks).

Remember that the set of tests provided is never complete. Your task is to write a function that solves the problem that is described for all valid arguments (i.e., all arguments that meet the restrictions given in the problem description). Passing the tests does not prove that your implementation is correct, but *failing any test proves that your code is wrong.*

Problem 1

A sequence of numbers is called *super-increasing* if and only if every number in the sequence is strictly greater than the sum of all numbers preceding it in the sequence. The first element in the sequence can be any number.

For example, the sequences `1,3,5,11,21` and `-2,1,2` are both super-increasing; the sequence `1,3,5,7,19` is increasing, but not super-increasing.

Write a function `super_increasing(seq)` that takes as argument a sequence of numbers and returns True if the sequence is super-increasing and False if it is not.

- You can assume that the argument is a non-empty sequence of numbers.
- You should not assume that values in the sequence are unique or increasing.
- You should not make any assumption about the type of the argument other than that it is a sequence type; likewise, you should not make any assumption about the type of the numbers in the sequence, other than that they are numbers.
- The function must not modify the argument sequence.
- The function must return a truth value (a value of type `bool`).

Files:

- Skeleton file: super_increasing.py

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `super_increasing` that has one parameter. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_super_increasing`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*

Problem 2

A closed interval of the real number line is defined by its lower and upper end points. Write a function `interval_intersection(lA, uA, lB, uB)` that returns the length of the intersection of two intervals A and B. Arguments `lA` and `uA` are the lower and upper end points of interval A, and `lB` and `uB` are the lower and upper end points of interval B. If the intervals do not intersect, the function should return 0.

For example, `interval_intersection(0, 3, 1, 5)` should return `2`, because the intersection of the two intervals [0,3] and [1,5] is [1,3], which has a length of 3 - 1 = 2.

- You can assume that the function's arguments are numbers, but NOT that they are integers, or positive.
- Your function must return a number.

Files:

- Skeleton file: interval_intersection.py

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `interval_intersection` that has four parameters. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_interval_intersection`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*

Problem 3

The integral of any function over a bounded interval [a,b] can be approximated by a sum. Each term in the sum approximates the integral over a small part of the interval with the area of a trapezoid that has two corners on the x-axis and two on the curve. For example, the following figure shows the curve x - x2 and two trapezoids: one over the interval [0.1, 0.4] and one over the interval [1.1, 1.4].

Let f(x) be the function that we want to integrate. The area of the trapezoid over the interval [x,x+d] is ((f(x) + f(x + d)) / 2) * d.

Write a function `approximate_intergral(lower, upper, nterms)` that calculates an approximation of the integral of the function f(x) = x3 (that is, x cubed) over a bounded interval using the trapezoid method. The three parameters of the function are the lower and upper bound of the interval, and the number of terms in the sum. The function should calculate the sum of `nterms` trapezoids of equal width.

- You can assume that `lower` is less than or equal to `upper`.
- You can assume that `nterms` is a positive integer.
- Your function should return a numeric value.

Example:

- `approximate_integral(0, 2, 2)` should return 5. The first trapezoid is over the interval [0,1] and has an area of 0.5 (((0 + 1) / 2) * 1); the second is over the interval [1,2] and has an area of 4.5 (((1 + 8) / 2) * 1).

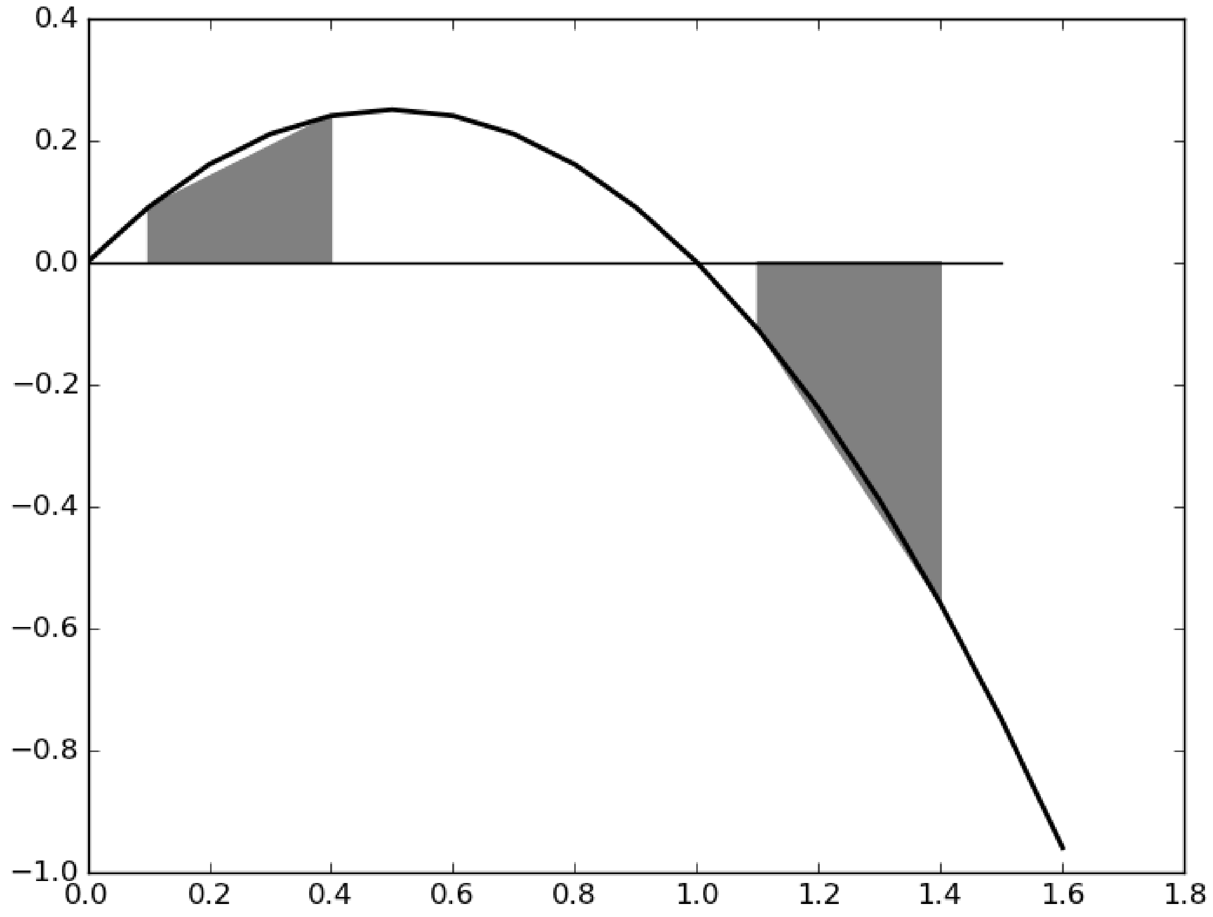

Figure 1: illustration of curve and two trapezoids

Files:

- Skeleton file: approximate_integral.py.

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `approximate_intergral` that has three parameters. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_approximate_integral`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*

Problem 4

Let's say that a *counting dictionary* is a dictionary in which all values (not keys) are positive integers. (In other words, it's the kind of dictionary you get when you use a dictionary to count repetitions of keys.) Given two counting dictionaries, `A` and `B`, the *difference between `A` and `B`* is another counting dictionary that contains the key-value pairs `(k, n)` where `n = A[k] - B[k]` for exactly those keys `k` such that `n > 0` or `k` appears only in `A`.

Write a function `count_dict_difference(A, B)` that takes as arguments two counting dictionaries and returns a new dictionary that is the difference between them.

- You can assume that both arguments are dictionaries, and the values stored in them are integers.
- You should make *no* assumption about the types of keys that can appear in the dictionaries.
- Your function *must not* modify either of the argument dictionaries.
- Your function should value of type `dict`.

Example:

For example if `A = {'s': 4, 'm': 1, 'p': 2, 'i': 4}` (that's the letter counts for the word 'mississippi') and `B = {'e': 1, 's': 3, 'm': 1, 'p': 1, 'i': 2, 't': 1}` (that's the letter counts for the word 'pessimist'), the difference of `A` and `B` is the dictionary `{'s' : 1, 'p' : 1, 'i' : 2}` (note how `'m'` is not in the difference because `A['m'] - B['m'] == 0`).

Files:

- Skeleton file: count_dict_difference.py

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `count_dict_difference` that has two parameters. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_count_dict_difference`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*

Problem 5

Write a function `remove_all(a_list, element)`, which removes *all* occurrences of `element` from `a_list`. You can assume that the first argument to the function, `a_list`, is a list. The function should not return any value (i.e., return `None`) but should modify the argument list. For example,

```
In [1]: my_list = [1,2,3,2,3,1,2]
In [2]: remove_all(my_list, 2)
In [3]: my_list
Out [3]: [1,3,3,1]
```

Files:

- Skeleton file: remove_all.py.

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `remove_all` that has two parameters. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_remove_all`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*

Problem 6

The *moving average* of a numeric sequence is a sequence of averages of subsequences of the original sequence; the subsequences are known as *windows* (or a *sliding window*), and the length of the subsequence used is called the *window size*. More precisely, given a sequence `S`, the moving average with window size `w` is sequence `A` such that the first element in `A` is the average of `w` consecutive elements in `S` starting with the first, the second element in `A` is the average of `w` consecutive elements in `S` starting with the second, and so on until the last element in `A`, which is the average of `w` consecutive elements in `S` ending with the last.

Example:

If `S = (2, 0, -2, 2)` and `w = 2`, the moving average is `[1, -1, 0]`. `1` is the average of `(2, 0)`, `-1` is the average of `(0, -2)`, and `0` is the average of `(-2, 2)`.

Write a function `moving_average(seq, wsize)` that computes and returns the moving average of `seq` with window size `wsize`.

- You should assume that the argument is a sequence of numbers (integer or floating point, or a mix of them).
- You should NOT make any assumption about the type of the sequence (e.g., it could be list, tuple or NumPy array).
- You can assume that `wsize` is an integer, at least 1, and less than or equal to the length of `seq`.
- Your function must return a sequence of numbers (this can be, for example, a list, a tuple or NumPy array).

Files:

- Skeleton file: moving_average.py

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `moving_average` that has two parameters. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_moving_average`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*

Problem 7

We say that a list that contains lists is *nested*. The result of *unnesting* (also known as "flattening") a nested list is a list that contains the same non-list elements as the nested list, in the same order, but in a single list.

For example, `[1, [2], [[3], [[4], 5]]]` is a nested list. The unnesting of this list is `[1, 2, 3, 4, 5]`.

Write a function called `unnest` that takes as argument a list, which may be nested, and returns the unnesting of the argument list.

- You should assume that the argument is a list, and that all elements in the list are either lists or some non-sequence type.
- Your function *must not* modify the argument list. It must return a value of type `list`.

Files:

- Skeleton file: unnest.py

You should write your solution into this file. Remember that:

- The file must contain only syntactically correct python code (and comments).
- Do not import any module that you do not use.
- You must define a function named `unnest` that has one parameter. You may also define additional functions if it helps you break down or solve the problem.

Using the testing function:

The skeleton code file includes a testing function, `test_unnest`, which will run some tests on your function and raise an error if any test fails. Passing the tests does not prove that your solution is correct. *Failing any test proves that your function is wrong.*