PE4: PE of 08/01/2019 (solutions)

Master in Informatics and Computing Engineering
Programming Fundamentals
Instance: 2018/2019

Some **important** information about this PE (Practical on computer evaluation):

- You have **90 minutes** to answer the 5 questions of the test
- **No collaboration** between students is allowed
- **It is forbidden** the presence on the table and the use of mobile phones or any other electronic devices
- The Python code that answers each question is saved in a file with **the name required in the question**
- **Before the time expires** you must *upload* a zip with the Python code of all your answers; you have **only one attempt** but you may upload the zip as many times as you wish; therefore, you should try the upload procedure at least 5 minutes before the time expires, to **guarantee you have one zip** with the answers to be graded
- Your are allowed to use the **Consultation Book** and the **Standard Library**, both in PDF, but all the remaining content will be hidden

1. Greatest common divisor

Write a Python function `gcd(a, b)` that given two numbers (`a` and `b` with `a > b`) computes the greatest common divisor (the largest number that divides both of them without leaving a remainder).

The Euclid's algorithm is an efficient method for computing the greatest common divisor of two numbers and works by iteratively replacing the larger of the two numbers by its remainder when divided by the smaller of the two; the algorithm stops when reaching a zero remainder.

Save the program in the file `gcd.py`

For example:

- `gcd(25, 5)` returns the integer `5`
- `gcd(21, 14)` returns the integer `7`
- `gcd(65, 26)` returns the integer `13`

Solution:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

2. Ackermann function

Write a Python function **`ackermann(m, n)`** that, given two non-negative integers `m` and `n`, computes the value of the Ackermann function A(m,n).

The Ackermann function can be described as follows:

- If m = 0, A(m,n) = n+1
- If m > 0 and n = 0, A(m,n) = A(m-1,1)
- else, A(m,n) = A(m-1, A(m, n-1))

The value of the function grows rapidly, even for small inputs, so be careful to test your implementation with small values of `m` and `n`.

Save the program in the file **`ackermann.py`**

For example:

- `ackermann(0, 0)` returns the integer `1`
- `ackermann(2, 1)` returns the integer `5`
- `ackermann(3, 4)` returns the integer `125`

Solution:

```
def ackermann(m, n):
    if m == 0:
        return n+1
    elif n == 0:
        return ackermann(m-1, 1)
    else:
        return ackermann(m-1, ackermann(m, n-1))
```

3. Histogram

Write a Python function `histogram(alist, bins)` that receives a `alist` of numbers and a tuple `bins` indicating how numbers should be divided in groups. The function returns the frequency distribution of the numbers according to the division by bins.

Given `alist=[1, 1, 1, 4, 5, 8, 10]` and `bins=(0, 3, 7, 12)`, then there is the following frequency distribution:

`bins`	frequency
[0, 3[	3
[3, 7[	2
[7, 12[	2

and, therefore, the function returns the list `[3, 2, 2]`.

Save the program in the file `histogram.py`

For example:

- `histogram([1, 1, 1, 4, 5, 8, 10], (0, 3, 7, 12))` returns the list `[3, 2, 2]`
- `histogram([0, 3, 4, 7, 8, 1, 5], (0, 3, 7, 12))` returns the list `[2, 3, 2]`
- `histogram([3, 0, 1, 5, 3, 2], (0, 3, 6))` returns the list `[3, 3]`

Solution:

```
def histogram(alist, bins):
    freqs = [0] * (len(bins)-1)
    for n in alist:
        j = 0
        while n >= bins[j]:
            j += 1
        freqs[j-1] += 1
    return freqs
```

4. Maximum depth

Write a function `maximum_depth(l)` that receives a list `l`, which can contain other lists, and returns what is the maximum depth in that list.

The depth corresponds to the number of sub-lists: `[]` has depth=1, `[[]]` has depth=2, `[[[]]]` has depth=3.

Save the program in the file `maximum_depth.py`

For example:

- `maximum_depth([[], [[]], [], [[]]])` returns the integer `3`
- `maximum_depth([[[], [], [[]]], [[]], [], [[]]])` returns the integer `4`
- `maximum_depth([[[], [], [[]]], [[[[]]]]])` returns the integer `5`

Solution:

```
def maximum_depth(lst):
    if lst == []:
        return 1
    maxd = 0
    for i in lst:
        d = maximum_depth(i)
        if d > maxd:
            maxd = d
    return maxd + 1
```

5. Sum zip generator

Write a **generator** function `sum_zip(functions, arguments)` that, given a list of `functions`, and a list of `arguments`, successively yields tuples with: (i) a list with the evaluation of each argument by each one of the functions, and (ii) the sum of the elements in such a list.

Notice that each function that is given accepts one and only one argument.

Save the program in the file `sum_zip.py`

For example:

-
  ```
  sum_zip([lambda x: x*2,
           lambda x: x**2,
           lambda x: -x],
          [-5, 10, 3])
  ```
  returns a generator that produces the list of tuples `[([-10, 25, 5], 20), ([20, 100, -10], 110), ([6, 9, -3], 12)]`

-
  ```
  sum_zip([lambda x: x*2,
           lambda x: x**2],
          [2, 3, 9])
  ```
  returns a generator that produces the list of tuples `[([4, 4], 8), ([6, 9], 15), ([18, 81], 99)]`
- `sum_zip([lambda x: x+2], [1])`
  returns a generator that produces the list of tuples `[([3], 3)]`

Solution:

```
def sum_zip(functions, arguments):
    for a in arguments:
        rl = [f(a) for f in functions]
        yield rl, sum(rl)
```

The end.

FPRO, 2018/19