Exam Computer Programming II 2022-10-28

Exam time: 08:00 – 13:00 (last entrance 08:20)

The exam takes place in the computer labs at Ångström according to announcement in Studium.

During the exam:

- You may ask the exam guards for help with login and installation problems during the exam. However, you can not count on others to solve technical problems, it is your own responsibility that it works.
- Have the Google-document (queue list) open dureing the exam:
  https://docs.google.com/spreadsheets/d/1AAK2InF0JAZP0lkhgGlrHBqI1dd9Voq2k0epjZ8o3Gc/edit
  (choose the tab "Exam 28/10 8" in the bottom)
  open during the exam. There we will write any corrections and clarifications. Major changes are also made with advertising in Studium and exam guards.

  There is also room for questions from students and answers to these from the teachers.

Submission:

- Submission of the exam takes place through uploading the Python files to the same location where the exam was retrieved in Studium (in the same way as a standard assignment). **Note that you must enter your name as a comment in the beginning of all files!**
- The exam consists of A- and B-tasks. A-tasks must work (submitted programs must be able to run and solve the task) to be approved. B-tasks can give "points" even if they do not solve the problem completely
- All submitted code must be executable.
- Make a zip file (not rar, 7z or something else) that contains all your files and upload it to Studium.

  If you do not know how to make zip files, you can upload the files directly in one upload. Drag and drop all the files at the same time to Studium. It is possible to upload files several times, make sure you upload ALL files again, and then it is the latest version that applies (they will be named `ma1-1.py`, `ma1-2.py` and so on).
  Do not forget to click "Submit"!
- If there is a problem with Studium you can send the zip file to sven-erik.ekstrom@it.uu.se. Ask an exam guard to do this together with you. You must ask them no later than 13:00:00.
- **No submissions after 13:00:00 will be accepted!**

Rules

- You may not collaborate with anyone else during the exam, neither physically nor electronically. To have e.g. an email or a chat client open at the time of writing will be reported as attempted cheating.
- You may not have headphones, look at videos, or listen to sound thru speakers.
- You may not copy code or text but must write your answers yourself.
- You may use the internet. If you download code or algorithms from the web, or if are heavily inspired by some code you find, you must provide the source with the full link (better to give too much citations than too little).
- You should write your solutions *in designated places* in the files `m1.py`, `m2a.py`, `m2b.py` , `m3.py`, and `m4.py`. You must keep the names of files, classes, methods and functions. Functions must be able to be called exactly in the way stated in the task. For tasks in module 1 and 3 it is allowed to add parameters with default values but the function or method must still be callable in the way stated in the task.
- You may not use packages other than those already imported into the files unless otherwise stated in the task. (The fact that a package is imported in *does not* mean that it needs to be used!)
- You can write and use helper functions.
- Before your exam is approved, you may need to explain and justify your answers orally to teachers (after the exam, once we have corrected the exam).

PLEASE NOTE THAT WE ARE OBLIGATED TO REPORT
ANY SUSPICION OF ILLEGAL COOPERATION OR COPYING
AS A POSSIBLE ATTEMPT TO CHEAT!

Components of the exam:

The exam is divided into four sections, each with two A and one B assignment corresponding to each of the four modules. Hence, there are a total 8 A-tasks and 4 B-tasks in total.

Grading requirements:

3: At least six A-tasks approved, where at least one task is approved for each module.

4: At least six A-tasks approved and either two B-tasks largely correct or the voluntary submission approved.

5: At least eight A-tasks approved and either all B-tasks or three B-tasks and the voluntary submission.

Note: we may curve the grading, and change the criteria above, so it is still worth to hand in even if you have not met grading criteria mentioned above.

Tasks related to module 1

The solution to this task should be written in the designated places in the file `m1.py`. The file also contains a `main` function that demonstrates the function of the code.

A1: Write a function `depth(arg)` that returns the depth of `arg`. The depth is defined as 0 if `arg` is not a list otherwise is the depth 1 plus the depth of the deepest of the elements in the list. Sublists should be handled with recursion.

Example:

```
    lists = (1, [], [[]], [1, 2, 3], [[1], 2, [[3]]],
             [[1, (1, [2])], [[[[2]]]], 3], ['[[', ']']])
    print(f'Result      Argument')
    for lst in lists:
        print(f'{depth(lst):3d} \t     {str(lst):35}')
```

Output:

```
Result      Argument
  0          1
  1          []
  2          [[]]
  1          [1, 2, 3]
  3          [[1], 2, [[3]]]
  5          [[1, (1, [2])], [[[[2]]]], 3]
  2          ['[[', ']']]
```

A2: Write the function `is_sorted(lst)` that returns `True` if the elements in the list `lst` are sorted in ascending order, otherwise `False`. If there are elements in the list that are not comparable in size the function should return `False`. The function must be implemented with *recursion* and must therefore not contain any iteration.

Example:

```
    args = ([], [1, 2], [1, 3, 2], [2, 3, 5, 4], ['a', 'ab', 'c'],
            [1,'a'], [0, False], [[1,2,2],[1,2,3]])
    print('Result      Argument')
    for a in args:
        print(f' {is_sorted(a)} \t      {str(a):35}')
```

Output:

```
Result       Argument
 True         []
 True         [1, 2]
 False        [1, 3, 2]
 False        [2, 3, 5, 4]
 True         ['a', 'ab', 'c']
 False        [1, 'a']
 True         [0, False]
 True         [[1, 2, 2], [1, 2, 3]]
```

B1: Given the function

```
def foo(n, m=9):
    c=[x for x in range(1,m)]
    def fie(n, c):
        if n == 0:
            return 1
        elif n < 0 or len(c) == 0:
            return 0
        else:
            return fie(n, c[1:]) + fie(n-c[0], c)
    return fie(n, c)
```

Investigate how the time of the call `foo(n)` depends on `n`. Estimate how long `fib(500)` would take. Assume that recursion depth does not cause any problem.

Place the code that you use in the `main` function and put the output from that code and the calculations in the comment field at the end of the file.

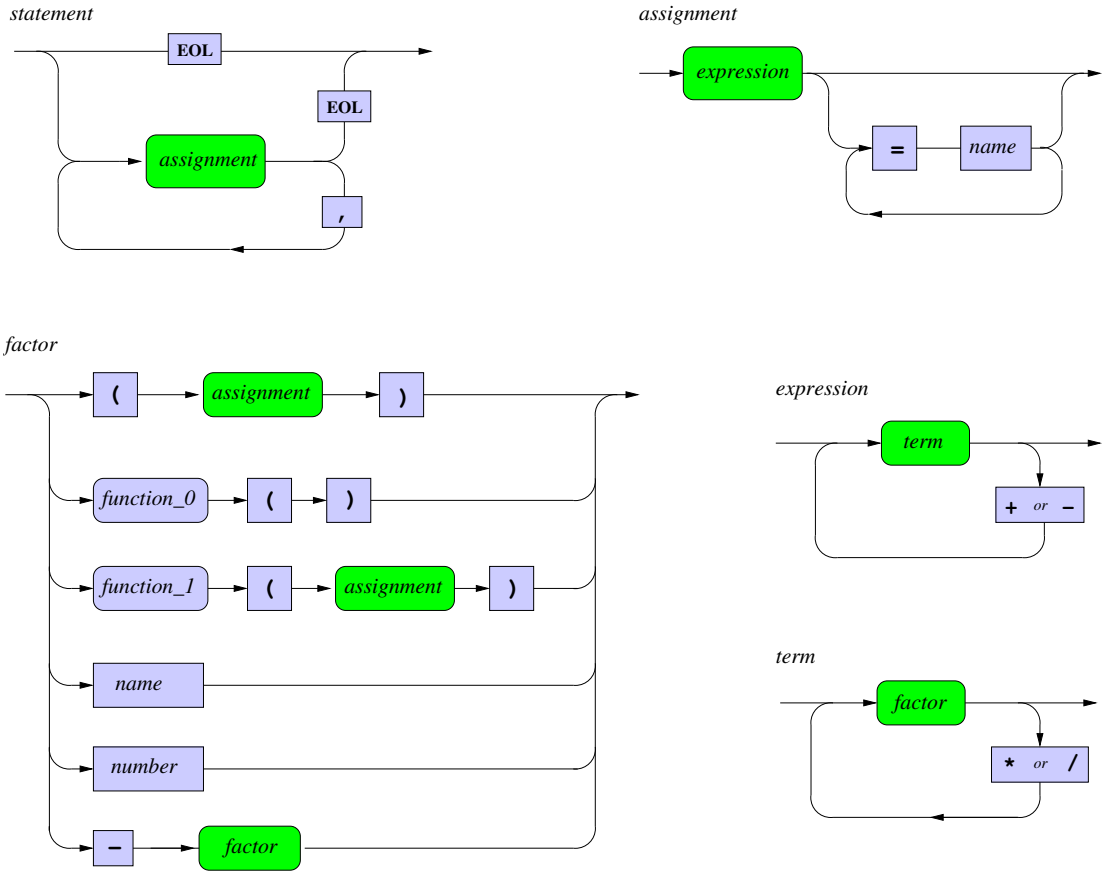
`Tip:` Time measurements always give varying results and if constants based on such are used to estimate times for large arguments, you can sometimes get *very* different estimates in repeated tests.

Tasks related to module 2

The downloaded code `m2a.py` with the initialization file `m2a_init.txt` implements a slightly simplified calculator similar to the one in assignment MA2. The file `m2b.py` and its initialization file `m2b_init.txt` is a variant to be used in task B2.

The `TokenizeWrapper` class, which is used by both variants, is in its own file.

The syntax of tasks A3 and A4 is described by the following diagram:



The program starts by reading from the file `m2a_init.txt` (which you downloaded). The file starts like this:

```
1 1 - (2 - 4) = x      # 3.0
2 x*4/2 + x            # 9.0
3 (1=x) + (2=y)*(3=z) # 7.0
```

The expected result is after the comment character `#`.

In the rest of `m2_init.txt` contains test cases for the tasks. They will not work properly properly until you have solved the tasks.

Note: If your program fails to read the input file, you can enter the appropriate test cases yourself!

Note: You may *not* change the parameter lists in the given code!

A3: According to the syntax diagrams, it should be possible to write several *assignments* separated by commas in one *statement*. Example:

```
Init : 1, 2, 3, 4, 5          # Last evaluated value is the value
5

Init : ans
5

Init : 1 = x, 2 = y, x + y    # Assignment performed successively
3

Init : 10=x 11=y  3           # Syntax error - no comma
*** Syntax error:  Expected comma or end of line
*** Error occurred at '11' just after 'x'

Init : x                      # x got a value
10

Init : (1, 2, 3)              # Not allowed
*** Syntax error:  Expected ')'
*** Error occurred at ',' just after '1'
```

Rewrite the code to work according to the syntax diagrams!

A4: In the given code, only functions with *one* argument are accepted. According to the syntax diagrams, it should be possible to have functions without arguments. Implement that in the code and add the function `random` which should return one random number (float) between 0 and 1 and `time` which should return a string with current date and time.

Hint: `random.random()` and `time.ctime()` provide the answers needed. Both `random` and `time` are imported in the downloaded code. Example:

```
Init : random()
0.0036297631025310473

Init : random(1)
*** Syntax error:  Expected ')'. This function has no arguments.
*** Error occurred at '1' just after '('

Init : time()
Sun Oct 23 10:21:21 2022

Init : time(1)
*** Syntax error:  Expected ')'. This function has no arguments.
*** Error occurred at '1' just after '('

Init : time
*** Syntax error:  Expected '(' after function name.
*** Error occurred at '' just after 'time'
```

B2: In the file `m2b.py` there is an embryo of another implementation of a calculator. In this version, the expressions are read and stored unevaluated in a binary tree. Each tree node represents either an operation (`+`, `-`, `*`, ...) or a constant or variable. The parser therefore does no calculations, but only builds the tree.

The task consists of writing the method `evaluate(self, variables)` in the class `Node` which calculates the value of the (sub-)tree in which the node is the root.

Example:

```
From file: 1 - (2 - 4)*(5-1)
Parsed   : 1-(2-4)*(5-1)
Evaluated: 9

From file: 23%5*(3-1)
Parsed   : 23%(5*(3-1))
Evaluated: 3

From file: 2**2**3
Parsed   : 2**(2**3)
Evaluated: 256

From file: ans*10
Parsed   : ans*10
Evaluated: 2560

From file: 17//4
Parsed   : 17//4
Evaluated: 4

From file: pi*(pi-2)
Parsed   : pi*(pi-2)
Evaluated: 3.5864190939097718

From file: 4%(6-2*3)
Parsed   : 4%(6-2*3)
*** Evaluation error:  Division by zero

From file: 2+3 = x       # Don't implement evaluation of this
Parsed   : 2+3=x
*** Evaluation error:  Undefined: =

From file: x+8           # thus x is undefined
Parsed   : x+8
*** Evaluation error:  Undefined: x
```

The operations must be stored in a dictionary. To evaluate new operators one must thus not having to change the code but just update lists and dictionaries.

The assignment operation (`=`) is included in the syntax handling, but you should not implement it. This means that your code does not need to be able to store new values in `variables` but only use things that are already there.

Tasks related to module 3

In this section you will work with the file `m3.py` which contains the classes `LinkedList` and `BST`.

There is also a `main` function with some small demonstration runs. These are not comprehensive tests, but you should write more yourself!

The `LinkedList.py` class contains code to handle *linked lists* of objects. The lists are kept sorted by the `insert` method, but the same value can appear multiple times (and of course next to each other).

The `BST` class contains code for standard binary search trees.

A5: The method `copy()` in the class BST should construct a copy of the tree. The method looks like this:

```
    def copy(self):

        def _copy(r):  # Task A6
            pass

        return BST(_copy(self.root))
```

Write the internal helper method `_copy`.

The copy must have its own nodes and be completely independent from the original.

A6: In the code there is the following function:

```
def build_list(n):
    llist = LinkedList()
    for x in range(3*n):
        llist.insert(x)
    return llist
```

How does the time depend on the parameter n? Answer with a Θ expression!

Estimate how long time the call `build_list(1000000)` would take on the computer you are using.

Place the code you use in the designated place in the `main` function and report printouts and reasoning in the "triple quota comment" at the end of the file!

B3: Write a class `LevelOrderIterator` that can be used to traverse a tree in level order, i.e. first the root, then the root's children, etc.

Examples of use:

```
    print('\nB3: LevelOrderIterator')
    bst = BST()
    print('Insertion order: ', end=' ')
    for x in [5, 3, 2, 4, 8, 10, 6, 1, 7, 9]:
        print(x, end=' ')
        bst.insert(x)
    print('\nSymmetric order: ', end=' ')
    for x in bst:
        print(x, end=' ')
    print('\nLevel order    : ', end=' ')
    loi = LevelOrderIterator(bst)
    for x in loi:
        print(x, end=' ')
    print()
```

Output:

```
B3: LevelOrderIterator
Insertion order:  5 3 2 4 8 10 6 1 7 9
Symmetric order:  1 2 3 4 5 6 7 8 9 10
Level order    :  5 3 8 2 4 6 10 1 7 9
```

Tasks related to module 4

No other Python modules/packages are to be used, than the ones that are explicitly permitted.

Allowed modules to use are `random`, `functools.reduce`, and any multiprocessing module you want.

A7: Modify the method `birthdays(n_people)` in `m4.py` such that the following is fulfilled:

- at least one higher order function, discussed in the course, should be used;
- create random birthdays for `n_people` (an integer) people (you can assume that there are only 365 days, leap years can be ignored, and the method `random.randint()` may be useful);
- return 0 if none on the `n_people` people share the same birthday;
- return 1 if at least two people share the same birthday (also return 1 if for example three people share the same birthday, or if two pairs share birthdays).

A8: This task is a continuation of task A7 and assumes that the method `birthdays(n_people)` works. If you have not done task A7, then you can use the method `birthdays_theoretical(n_people)`, found in `m4.py`, instead. Read the description of what the method does in task A7. In the rest of the description of the current task you can just switch `birthdays` to `birthdays_theoretical`.

In this task you will modify the method `print_birthday_statistics(n_peoples, n_sample,n_processes)` in `ma4.py`, so that is prints statistics given by the method `birthdays`.

We start with an an example of a print with a correct call

```
print_birthday_statistics(range(15,30),10000,4)
```

This example is used as a test in the bottom of `m4.py`. Note that the output does not have to match exactly, and that the values will vary for every run.

```
>>> print_birthday_statistics(range(15,30),10000,4)

15: 0.2547
16: 0.2817
17: 0.3111
18: 0.3424
19: 0.3867
20: 0.4127
21: 0.4467
22: 0.475
23: 0.497
24: 0.5431
25: 0.5716
26: 0.5962
27: 0.6296
28: 0.6599
29: 0.6828
```

- The first argument is the list `n_peoples`, and in this case it is `range(15,30)`, i.e., $[15, 16, \ldots, 29]$, the different number of people that for each we will estimate the probability for which two or more people share birthdays.
- The second argument is `n_samples`, and in this case it is `10000`. The total number of times we call `birthdays` for each number `n_people` given in the first argument (e.g., with the call `print_birthday_statistics([10,17],100,4)`, one should call `birthdays(10)` and `birthdays(17)` 100 times each).
- The third argument is `n_processes`, and in this case it is `4`. This is the number of processes/threads that you should use when you call `birthdays`. You can use any module you want for this.
- For each `n_people` (the elements in `n_peoples`) you should present, like in the example above, the ratio of calls to `birthdays` where two or more people share birthday with the total number of calls (i.e., how big is the chance that two people or more have the same birthday if you have a group of `n_people` people).
- You can assume that `n_samples` is divisible by `n_processes`. The code must work with other arguments than the test in the end of `m4.py`. For simpler grading we want the method `print_birthday_statistics` to print out all the statistics in the terminal, not return anything. The parallelization does not have to make the code faster, only work.

B4: Among the files you downloaded for the exam is `customers.json`. It is a so-called JSON file, a file format used for simpler structured data (you can open the file in an editor to see the structure, but it is not needed to solve the task).

The file `customers.json` contains a fictitious data base over customers of a company, where every customer has different information/attributes stored about them; there are 112 customers in the data base.

In the file `m4.py` there is an example function `get_name(index)` that retrieves the attribute `name` for a customer with a certain `index` $(0, \ldots, 111)$:

```
import json
def get_name(index):
    with open('customers.json') as f:
        data = json.load(f)
        return data[index]['name']
```

For example, calling `get_name(0)` returns the name `Laura Pitts` and `get_name(1)` returns the name `Battle Mcneil`.

Modify the method `print_favoriteFruits_per_gender(jsonfile, n, n_processes)` in `m4.py` so that

- it prints the information regarding **favorite fruit** (`favoriteFruit`) for all the customers, according to the following:

Count, separated by the **gender** (`gender` that is here assumed to be `female` or `male`) of the customers, how many customers have respective favorite fruit. A correct printing of the given data base, using the call `print_favoriteFruits_per_gender('customers.json',112,5)`, can be of the form:

```
>>> print_favoriteFruits_per_gender('customers.json',112,5)

-------
female:
-------
banana: 20
apple: 19
strawberry: 17
orange: 1
pear: 1
-------
male:
-------
banana: 16
apple: 13
strawberry: 23
orange: 2
-------
```

- the ordering of the printing and exact formatting does not matter. The important thing is the correct counting of favorite fruits;
- the fruit types should not be hard coded, but be automatically be found for any similar data base;
- the method `print_favoriteFruits_per_gender(jsonfile, n, n_processes)` has `jsonfile`, `n`, and `n_processes` as arguments, that is, the methods should retrieve the information form the data base `jsonfile` that has `n` customers in parallel, divided on `n_processes` processes/threads. You can not assume that the number of customers `n` (112 in `customers.json`) is evenly divisible by `n_processes` (for example, 5), but the answer should be correct (the total number of favorite fruits should be 112 for `customers.json`);
- you can use any parallelization module that you want, but beyond that you should not import any modules of packages except `json` (which is included in Python so no need to install it).
- for simpler grading we want the method `print_favoriteFruits_per_gender(jsonfile, n, n_processes)` to print out all the statistics in the terminal, not return anything. The parallelization does not have to make the code faster, only work.