Exam Computer Programming II 2024-10-29

Exam time 14:00 – 19:00

Downloading the exam files

You download the files for the exam from **one** of the links in Inspera called 'Files' (same contents in all of them, so if one works for you, no need to look at the others).

In case you have issues with opening these `.py` files, we also provide the same content in `.pdf` (only allowed extension) files.

How to work with the exam

- The exam is divided into four sections with tasks corresponding to the four modules in the course. **Every section corresponds to *one* copy-paste (ctrl-c ctrl-v) into an answer-box in Inspera.**
- The exam contains A and B tasks. Tasks must work (submitted programs must be able to run and solve the task) to be approved.
- Write your solutions *in the places indicated* in the files `m1.py`, `m3.py` and `m4.py`. In `m2.py` you have to find yourself where the changes and additions should be made.
- Run and debug your codes in any of three editors available (bottom left).
- Make a 'backup' of your code regularly by copy-pasting (ctrl-c ctrl-v) your code into the solution boxes in Inspera. It has happened that the online editors have crashed during exams. You can paste you solutions any number of times into Inspera. Save that too!

Rules

- You may not collaborate with anyone else (or use AI resources such as ChatGPT or Copilot) 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. It is allowed to ask assistance where a specific symbols are located on the keyboard.
- You must keep names of files, classes, methods, and functions. Functions must be able to be called exactly as stated in the task.
- You may not have headphones, look at videos, or listen to sound thru speakers.
- Internet access is limited to the python interpreters, python documentation, and links to the files.
- You may not use packages other than those already imported in the files unless otherwise stated in the task.
- You may write and use help functions.

Submission You paste the whole files (`m1.py`, `m2.py`, `m3.py`, `m4.py`) in the respective answer boxes in Inspera.

Grade requirements:

3: At least four A tasks passed, of which at least one task passed in each module.

4: At least five A tasks passed and two B tasks correct.

5: At least five A tasks passed and three B tasks correct.

Note that we *can* lower the grade requirements, so it is worth submitting the exam even if you do not strictly meet the requirements stated above.

Tasks in connection with module 1

The solutions to the tasks must be written in designated places in the file `m1.py`. The entire `m1.py` must then be pasted into the corresponding Inspera box when submitting.

A1: The function below is only useful for small values of the argument `n`. Write a new version of the function that can handle large values on `n`. You may use any technique as long as it calculates the same thing as the given one.

```
def A1(n): """ A1: complete the function """
    if n < 4:
        return 1
    else:
        return 2*A1(n-1) - 4*A1(n-2) + 3*A1(n-3) -A1(n-4)
```

Example: the code

```
print(A1(10))
print(A1(90))
print(A1(200))
```

should produce the following output

```
-21
-1100087778366101931
-107168651819712326877926895128666735145224
```

A2: Palindrome is a word that reads the same backwards and forwards. The examples are 'madam' or '22/11/22'. Write a **recursive** function `A2` that returns True if an input string is a palindrome and False otherwise.

Example: the code

```
print(A2("racecar"))
print(A2("Realisationsvinstbeskattning"))
print(A2("madam"))
```

should produce the following output

```
True
False
True
```

B1: Write a **recursive** function `B1(lst)` that creates and returns two lists of the elements in `lst` where the first consists of the elements at index 0, 2, 4, ... and the other of the elements at position 1, 3, 5, ...

Example: The code

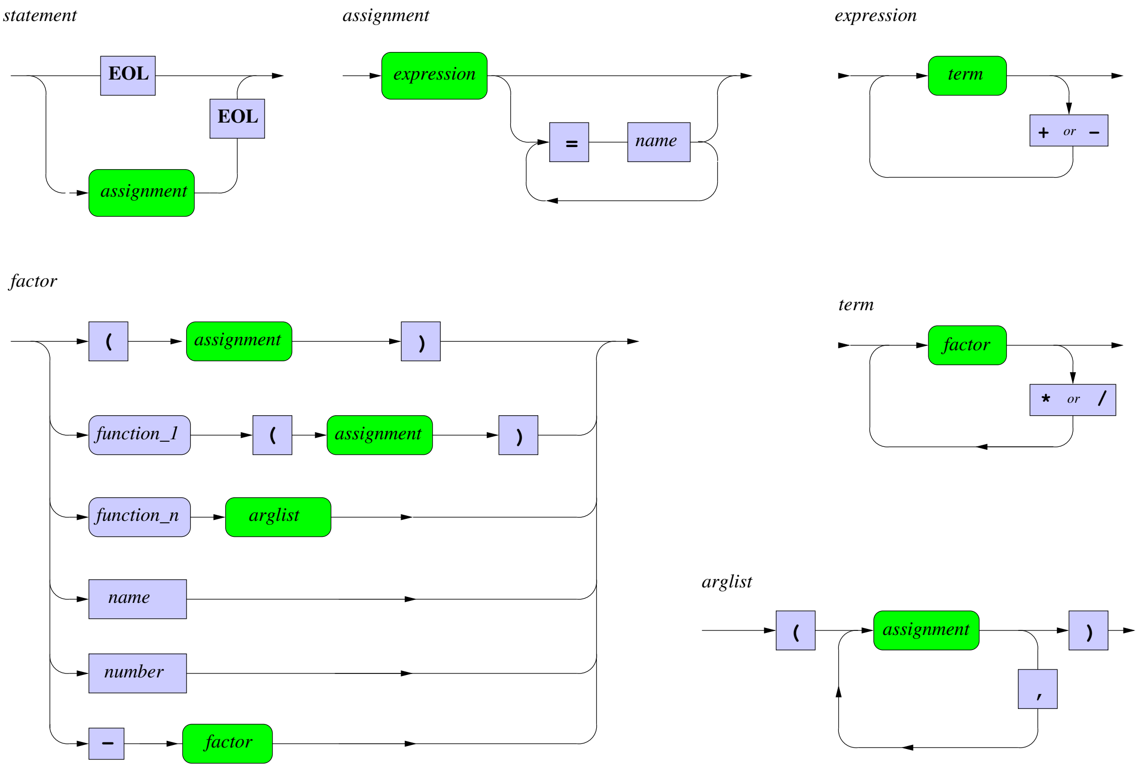
```
lists = ([],
         [0],
         [0, 1],
         [0, 1, 2],
         [0, 1, 2, 3],
         [0, 1, 2, 3, 4]
        )
for lst in lists:
    print(f'{str(lst):15s} \t {B1(lst)}')
```

should produce the following output

```
[]                         ([], [])
[0]                        ([0], [])
[0, 1]                     ([0], [1])
[0, 1, 2]                  ([0, 2], [1])
[0, 1, 2, 3]               ([0, 2], [1, 3])
[0, 1, 2, 3, 4]            ([0, 2, 4], [1, 3])
```

Tasks in connection with module 2

A3: The given file `m2.py` implements a calculator similar to the one in the second assignment. The file also contains the tokenizer. The syntax of the expressions is described by the following diagram:



In the given calculator there is the function `log` with one argument that calculates and returns the natural logarithm. Refactor the code so that `log` can take two arguments where the second argument is the base of the logarithm. If only one argument is given the natural logarithm. should be returned.

Example:

```
Input : log (10)
Result: 2.302585092994046
Input : log (2 ,10)
Result: 0.30102999566398114
Input : log (100 ,10)
Result: 2.0
Input : log (49 ,7)
Result: 2.0
Input : log (491 ,2.5)
Result: 6.762530616369424
Input : log (2 ,3 ,4)
*** Evaluation error: too many arguments to log: (2 ,3 ,4)
Input : log (0)
*** Evaluation error: Illegal arg to log: 0.0. Must be > 0
Input : log(2,-3)
*** Evaluation error: Illegal base to log: -3.0. Must be > 0
```

Note: The syntax diagrams should still be valid

Tip: This `log` should therefore work exactly like `math.log` except that it should throw `EvaluationError` on incorrect arguments.

Tasks in connection with module 3

A4: Write the `append` method in the `LinkedList` class that inserts a new node with the specified content as the last node in the list. The method should return the length of the extended list.

Example: the code

```
lst = LinkedList()
    print(lst)
    for i in [5,2,10,5]:
        l=lst.append(i)
        print(lst, l)
```

should produce the following output

```
()
(5) 1
(5, 2) 2
(5, 2, 10) 3
(5, 2, 10, 5) 4
```

A5: Write the method `sum_level(self, level)` in the class `BST` that returns the sum of the keys found at the level `level`. The level of the root is defined as 0, the level of the child of the root as 1, etc. You can assume that the keys are "summable".

Example: The code

```
inserts = (5, 8, 1, 3, 7, 2, 6, 9)
print(f'Inserted keys: {inserts}')
tree = BST(inserts)
print('Level   Sum of keys')
for level in (0, 1, 2, 3, 4):
    result = tree.sum_level(level)
    print(f'{level:3d} {result:10d}')
```

should produce the following output

```
Inserted keys: (5, 8, 1, 3, 7, 2, 6, 9)
Level   Sum of keys
  0          5
  1          9
  2         19
  3          8
  4          0
```

B2: In the given code for `BST` there is a `count` field in the tree nodes which is meant to store the number of nodes in the subtree that has that node as root.

Example:

1. In a tree with only one node, the value should be 1 in the root's `count` field.
2. In a tree with 2 nodes, the root should have 2 and the child 1 in the `count` fields.
3. In a tree with 3 nodes, the root must have 3 and the children either both have 1 or one 2 and the other 1 in the `count` fields depending on the shape.
4. All leaves must have 1 in the `count` fields.

Write a new method `insert(self, x)` that maintains the `count` fields as above.

Example: The code

```
inserts = (10, 5, 1, 7, 20, 30, 15, 17, 12, 2)
print(f'Inserted keys: {inserts}')
tree = BST()
for x in inserts:
    tree.insert(x)
    print(f'After inserting {x}: {repr(tree)}')
```

should produce the following output

```
Inserted keys: (10, 5, 1, 7, 20, 30, 15, 17, 12, 2)
After inserting 10: <(10, 1)>
After inserting 5: <(5, 1), (10, 2)>
After inserting 1: <(1, 1), (5, 2), (10, 3)>
After inserting 7: <(1, 1), (5, 3), (7, 1), (10, 4)>
After inserting 20: <(1, 1), (5, 3), (7, 1), (10, 5), (20, 1)>
After inserting 30: <(1, 1), (5, 3), (7, 1), (10, 6), (20, 2), (30, 1)>
After inserting 15: <(1, 1), (5, 3), (7, 1), (10, 7), (15, 1), (20, 3), (30, 1)>
After inserting 17: <(1, 1), (5, 3), (7, 1), (10, 8), (15, 2), (17, 1), (20, 4), (30, 1)>
After inserting 12: <(1, 1), (5, 3), (7, 1), (10, 9), (12, 1), (15, 3), (17, 1), (20, 5), (30, 1)>
```

The code requires only *one* descent in the tree.

Note: The `repr` method works like `str` but the nodes appears as a pair with the `key` and `count` fields.

Tasks in connection with module 4

A6: Write **a one-line** list comprehension to return all even elements of a given list.

Example: the code

```
numbers = [1, 3, 5]
print(f"Even numbers: {A6(numbers)}")
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 99]
print(f"Even numbers: {A6(numbers)}")
```

should produce the following output

```
Even numbers: []
Even numbers: [2,4,6,8,10]
```

A7: A function `A7(n)` gives the sum of squared $1^2 + 2^2 + ... + n^2$. Write the function `A7(n)` that uses at least three concepts you learned in the last course module, e.g., a lambda function and higher order functions such as map and reduce. Leave your response (code) in `m4.py` and a brief comment-description concerning concepts that you applied.

Tip: Output of `A7(n)` should be the same as of `A7_tester(n)`.

Example: the code

```
n=4
print(f"A7({n}): {A7(n)}")
print(f"A7_tester({n}): {A7_tester(n)}")
n=10
print(f"A7({n}): {A7(n)}")
print(f"A7_tester({n}): {A7_tester(n)}")
```

should produce the following output

```
A7(4): 30
A7_tester(4): 30.0
A7(10): 385
A7_tester(10): 385.0
```

B3: In `m4.py`, there is a function `random_matrices(m,n,p)` that creates two matrices A and B. The size of A is $[m, n]$ and the size of B is $[n, p]$.

In matrix multiplication $C = AB$, an element in C, c_{ij}, is a dot product of ith row of A and jth column of B.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Write matrix multiplication function `multiplication` in a parallel manner using at least two processes/ threads.

Example: The code

```
A, B= random_matrices(4,3,2)
C = multiplication(C)
print(A)
print(B)
print(C)
```

produces the output

```
[[8, 2, 2], [3, 1, 9], [4, 3, 2], [2, 3, 9]]
[[6, 2], [9, 4], [8, 7]]
[[82,  38], [99,  73], [67,  34], [111,  79]]
```

since

$$
\begin{aligned}
8 \cdot 6 + 2 \cdot 9 + 2 \cdot 8 &= 82 \\
8 \cdot 2 + 2 \cdot 4 + 2 \cdot 7 &= 38 \\
3 \cdot 6 + 1 \cdot 9 + 9 \cdot 8 &= 99 \\
\dots
\end{aligned}
$$

Note: A and B are random, you will have different values. The package `concurrent` is imported in `m4.py`. You can use other known packages for multiprocessing.

B4: An ordinary deck of cards consist of 52 cards (no jokers), divided into 4 colors (hearts ♥, diamonds ♦, clubs ♣, and spades ♠). Hence, there are 13 cards of different values of each color (1-10 -, jack , queen , king).

In poker you get a "hand", that is, 5 random cards from the deck of cards. In Table 1 you can see the different hands, and their theoretical chances.

Table 1: Poker hands and their theoretical chances.

Name	Description	Example	Theoretical chance
Nothing	None of the below		50.1177%
One pair	Two cards have the same value		42.2569%
Two pairs	Two pairs in the same hand		4.7539%
Triple	Three cards have the same value		2.1128%
Straight	Five consecutive cards (not the same color) Ace can be counted as the card before two, or after the King Valid: and		0.3925%
Flush	Five cards of the same color (not consecutive)		0.1965%
Full house	Triple and a pair		0.1441%
Quadruple	Four cards have the same value		0.02401%
Straight flush	Straight and also Flush		0.00139%
Royal straight flush	As Straight flush and smallest value is 10 (Ace comes after King)		0.000154%

In this task you need to parallelize a simulation of the theoretical chances in Table 1 for all types of poker hands. Do not forget "Nothing", and that Ace can come before a Two or after a King in a Straight (Valid: "poker hands" and).

- Modify `B4(n, n_processes)` in `m4.py`; `n` is the number of random poker hands that is used for the statistics; the higher `n` is, the closer to the theoretical chances in Table 1 one should be. The variable `n_processes` is the number of parallel processes that the computations should be done with. You can not assume that `n` is divisible by `n_processes`.

- You can use any way to represent unique cards; one way can be to identify every unique card as the values 1–52 (for example, 1-13 hearts, 14-26 diamonds, 27-39 clubs, and 40-52 spades). Ace is then $\{1, 14, 27, 40\}$.

- You can import any module/package for parallelization as you want. You can import anything from `random` and any parallelisation module/package you like. It is good to use as many higher order functions as possible. No other Python modules or packages, than the ones explicitly allowed, may be used to solve the problem.

- **Example**: Calling the function `B4(10000, 4)` may produce the following output in terminal.

  ```
  none : 0.501
  one_pair : 0.4244
  two_pairs : 0.0468
  triples : 0.0191
  straight : 0.0034
  flush : 0.0027
  full_house : 0.0021
  quadruple : 0.0005
  ```

- **Note:** With the randomization you can not be sure that all poker hands will be present in the simulation. For example, above there is no straight flush or royal straight flush.

- No other Python modules or packages, than the ones explicitly allowed, may be used to solve the problem. You can import anything from random and any parallelization module/package you like. It is good to use as many higher order functions as possible.

- **Tip:** `random.shuffle()` may be useful.