DAT516 Exam Instructions

Write your answers in Inspera. You can of course use separate sheets of paper to draft and experiment, but only the answers in Inspera will be graded. The answers can and should be short.

Python interpreters such as Self Practice are not available. Hence you have to reason about the answers on your own. Using pencil and paper is a good aid for this.

You can also have with you an A4 paper, hand-written on both sides, with whatever information you want to have available

You will need 24 points out of 60 in this exam to get accepted with grade 3, 36 points for grade 4, and 48 points for grade 5. The exam grade will also be the final grade for the course.

If used as a re-exam for DAT515 or DIT515 from earlier years, the final grade of the course will be the grade from your labs, provided that you get at least 24 points in this exam.

Python Syntax

For your reference: the syntax of (the relevant parts of) Python

```
<stm> ::= <decorator>* class <name> (<name>,*)?: <block>
    | <decorator>* def <name> (<arg>,*): <block>
    | import <name> <asname>?
    | from <name> import <imports>
    | <exp>,* = <exp>,*
    | <exp> <assignop> <exp>
    | for <name> in <exp>: <block>
    | <exp>
    | return <exp>,*
    | yield <exp>,*
    | if <exp>: <block> <elses>?
    | while <exp>: <block>
    | pass
    | break
    | continue
    | try: <block> <except>* <elses> <finally>?
    | assert <exp> ,<exp>?
    | raise <name>
    | with <exp> as <name>: <block>
<decorator> ::= @ <exp>
<asname>    ::= as <name>
imports     ::= * | <name>,*
<elses>     ::= <elif>* else: <block>
<elif>      ::= elif exp: <block>
<except>    ::= except <name>: <block>
<finally>   ::= finally: <block>
<block>     ::= <stm> <stm>*
<exp> ::= <exp> <op> <exp>
    | <name>.?<name>(<arg>,*)
    | <literal>
    | <name>
    | ( <exp>,* )
    | [ <exp>,* ]
    | { <exp>,* }
    | <exp>[exp]
    | <exp>[<slice>,*]
    | lambda <name>*: <exp>
    | { <keyvalue>,* }
    | ( <exp> for <name> in <exp> <cond>? )
    | [ <exp> for <name> in <exp> <cond>? ]
    | { <exp> for <name> in <exp> <cond>? }
    | { <exp>: <exp> for <name> in <exp> <cond>? }
    | - <exp>
```

| ~ <exp>
| not <exp>
<keyvalue> ::= <exp>: <exp>
<arg> ::= <name>
| <name> = <exp>
| *<name>
| **<name>
<cond> ::= if <exp>
<op> ::= + | - | * | ** | / | // | % | @ | == | > | >= | < | <= | != | in | not in | and | or
| & | | | ^ | << | >> | :=
<assignop> ::= += | -= | *= | /= | %= | &= | |= | ^=
<slice> ::= <exp>? :<exp>? <step>?
<step> ::= :<exp>?

1 Question 1

Question 1 (20 p). What is the value and its type of the following expressions? Remember that None is also a value! It can also happen that the expression contains an error. In that case, indicate the kind of error, one of SyntaxError, TypeError, AttributeError, ZeroDivisionError, KeyError, NameError, IndexError, and explain (in your own words) the reason of the error.

{1, 2, 3}.add(3)

(Value and Type) or (Error and Explanation)

{1, 2, 3} + {3}

(Value and Type) or (Error and Explanation)

set(1, 2)

(Value and Type) or (Error and Explanation)

[len(range(n)) for n in range(4)]

(Value and Type) or (Error and Explanation)

range(8)[-2]

(Value and Type) or (Error and Explanation)

{} is not {}

(Value and Type) or (Error and Explanation)

(lambda x, y: x(y))(lambda x: x, 8)

(Value and Type) or (Error and Explanation)

{2%x: x for x in range(1, 100)}

(Value and Type) or (Error and Explanation)

len([2%x: x for x in range(100)])

(Value and Type) or (Error and Explanation)

sum(range(1, 101))

(Value and Type) or (Error and Explanation)

Totalpoäng: 20

2 Question 2

Question 2 (10 p). In Lab 1 of this course, we built dictionaries of tram stops, tram lines, and transition times. They were collected into a single dictionary of the following shape:

```
tramnetwork = {
   "stops": {
      "Östra Sjukhuset": {
         "lat": 57.7224618,
         "lon": 12.0478166
      },
   # the positions of every stop
   },
   "lines": {
      "1": [
         "Östra Sjukhuset",
         "Tingvallsvägen",
         # and the rest of the stops along line 1
         ],
   # the sequence of stops on every line
   },
   "times": {
      "Östra Sjukhuset": {},
      "Tingvallsvägen": {
         "Östra Sjukhuset": 1
      },
   # the times from each stop to its alphabetically later neighbours
   }
 }
```

Using this definition, write a Python expression whose value is the set of those stops that are located south of "Chalmers", in the sense that their latitude is less than the latitude (lat) of Chalmers. All stop names are not shown in the fragment above, but you can assume that appear in the complete data. Notice: the answer must be an expression, not a statement or a sequence of statements.

Using the same definition, write a Python expression whose value is the set of those stop names that are neighbours of "Järntorget". Notice: the answer must be an expression, not a statement or a sequence of statements.

Totalpoäng: 10

3 Question 3

Question 3 (10 p). The following class defines undirected graphs in a way similar to Lab 2. A graph is initialized with an empty adjacency dictionary, which is then built up by adding edges. The dictionary is intended to give, for every vertex, the set of all its neighbors, but here in a non-redundant way where each neighbour relation is stored only once . For isolated vertices, this set is empty.

```
class Graph:
    def __init__(self):
        self.adjdict = {}
    def add_edge(self, a, b):
        "store new edge in the adjacency dict"
        if a <= b:
            self.adjdict[a] = self.adjdict.get(a, set())
            self.adjdict[a].add(b)
        else:
            self.adjdict[b] = self.adjdict.get(b, set())
            self.adjdict[b].add(a)

    def edges(self):
        "all edges but only in one direction"
        return {(a, b) for a, bs in self.adjdict.items() for b in bs}

    def remove_vertex(self, a):
        "remove vertex and all the edges that contain it"
        if a in self.adjdict:
            self.adjdict.pop(a)
```

The following piece of code builds a Graph and prints information about it. Show what is printed:

```
G = Graph()
G.add_edge(1, 2)
G.add_edge(3, 2)
G.add_edge(3, 4)
G.add_edge(1, 3)
G.add_edge(5, 3)
```

print(G.edges())

G.remove_vertex(3)

print(G.edges())

But watch out: something in these results is not correct. This is because there is a bug in some of the methods. Show a corrected version of that method:

Totalpoäng: 10

4 Question 4

Question 4 (10 p). Trees are a special case of graph, satisfying the following conditions:

- **edges are directed, that is, (a, b) is different from (b, a) - and, as the other conditions imply, only one direction can ever be included in the same graph**
- **exactly one vertex is the root, which means that there is no edge (a, root)**
- **all other vertices b have exactly one parent a; that is, there is exactly one edge (a, b) for every vertex b except the root**

You can test your understanding of trees by drawing on a paper the tree with the edges (1, 2), (1, 3), (2, 4), (2, 5), (5, 6). You will then see that it indeed becomes a branching, tree-like structure. The root is vertex 1.

Your task is to define a class Tree as a subclass of Graph from Question 3. You should do this with as few method overrides as possible. You can assume that the bug of Question 3 has been fixed and the class definition is right, even if you did not manage to fix the bug yourself.

Here are a couple of hints:

- **the** __init__() **method should take the root vertex as argument; otherwise, there would be no way to add more vertices**
- **the** add_edge() **method should ignore additions that violate the conditions: it should add (a, b) only if a is already in the graph and b is not**
- **the** remove_vertex() **method needs to remove all vertices that are reachable from the removed vertex by following edges, because otherwise some of those vertices would end up being without parents. This is the trickiest part of the question.**

Write your definition of the class here:

class Tree(Graph):

Show the adjacency dictionary resulting from the following statements and using your code:

```
T = Tree(1)
T.add_edge(1, 2)
T.add_edge(1, 3)
T.add_edge(2, 4)
T.add_edge(3, 4)
T.add_edge(5, 6)
T.add_edge(2, 5)
```

print(T.adjdict)

Show what happens after the removal of vertex 2. You can get full points (3p) even if your definition of remove_vertex() is not completely correct.

```
T.remove_vertex(2)
```

print(T.adjdict)

Totalpoäng: 10

5 Question 5

Question 5 (10 p). Decorators are functions that take other functions as arguments and modify them, returning the modified function. The following is an example:

```
def my_decorator(f):
    def my_f(*args):
        val = f(*args)
        if isinstance(val, int):
            return val * val
        else:
            return val
    return my_f
```

The application of a decorator can be expressed concisely with the following syntax:

```
@my_decorator
def square(n):
    return n*n
```

The effect of this is that any call of square(n) **becomes the call** (my_decorator(square))(n)**.**

Your first task is to show the output of the following expression:
print(square(3))

Secondly, given another decorated function,

```
@my_decorator
def full_name(firstname, familyname):
    return f'{firstname} {familyname}'
```

also show the output of the following expression:
print(full_name('Jane', 'Austen'))

Your third task is to define a decorator function repeat() **such that, given**

```
@repeat
def hello(x):
    print('hello', x)
```

the call hello('World') **prints**

```
hello World
hello World
```

hello World
hello World

Your answer, definition of the repeat() function:

Totalpoäng: 10