**Re-exam in Advanced Programming in Python (DAT515/DIT515)**

Chalmers University of Technology and University of Gothenburg
14 March 2024, 14:00 to 18:00, Johanneberg
Examiner: Aarne Ranta aarne@chalmers.se
Tel. 1082, mobile 0729 74 47 80

Write your answers directly below the questions. You can of course use separate sheets of paper to draft and experiment, but only the question papers will be graded. This reflects the fact that the answers can and should be short.

You will need 15 points out of 30 in questions 1-5 of this exam to get grade 3, 20 points for grade 4, and 25 points for grade 5. The exact grade from this exam is, however, not relevant for the final grade for the course. Your lab grade will be your final grade for the course, but it requires that you pass this exam (or another re-exam later) with at least grade 3.

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>
      |   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>?
```

**Question 1** (6 p). What is the *value* and the *type* of the following expressions? Remember that **None** is also a value!

- `(lambda x, y: y**x)(5, 2)`

  **Value:**

  **Type:**

- `[n % n for n in range(1, 4)]`

  **Value:**

  **Type:**

- `{max(n, 5) for n in range(10)}`

  **Value:**

  **Type:**

**Question 2** (6 p). Evaluating the following expressions raises errors. Which error is raised in each case? Use one of `TypeError`, `AttributeError`, `ZeroDivisionError`, `KeyError`, `NameError`, `IndexError`, and explain (in your own words) why.

- `{1, 2, 3}.append(2)`

  **Error:**

  **Reason:**

- `{1, 2, 3}.add([])`

  **Error:**

  **Reason:**

- `[1, 2, 3].append(4, 5)`

  **Error:**

  **Reason:**

**Question 3** (6 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 for a dictionary that to each tram stop assigns the set of lines that run through it. *Notice: the answer must be an expression, not a statement or a sequence of statements.*

**Answer:**

Assuming that the dictionary of the previous question has the name **stoplines**, write an expression whose value is the name of the tram stop that has the largest number of tram lines passing through it. Your solution can refer to **stoplines** even if you did not manage to define it properly.

**Answer:**

**Question 4** (6 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 each vertex, the set of it neighbours, where each neighbour is a vertex to which the

```
class Graph:
    def __init__(self):
        self.adjdict = {}

    def add_edge(self, a, b):
        "add b as neighbour of a and a as neighbour of b"
        self.adjdict[a] = self.adjdict.get(a, set())
        self.adjdict[a].add(b)
        self.adjdict[b] = self.adjdict.get(b, set())
        self.adjdict[b].add(a)

    def get_edges(self):
        "return the set of edges, each of them only in one direction"
        edges = set()
        for a in self.adjdict:
            for b in self.adjdict[a]:
                edges.add((a, b))
```

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(2, 3)
G.add_edge(2, 2)

print(G.adjdict) # prints:

print(G.get_edges()) # prints:
```

But watch out: the result of `get_edges()` is not correct. This is because there are bugs in the `get_edges()` method. On the line above, you should show the buggy result exactly as produced by the buggy code. But now, show the code needed to correct this method:

**Answer:**

**Question 5** (6 p). **Connected graphs** are graphs where one can get from any vertex to any other one by following edges. The Graph class defined in Question 4 allows graphs that are not connected: for example, adding the edges (1, 2) and (3, 4) leads to an unconnected graph.

The following subclass is otherwise like Graph, but it forbids unconnected graphs by simply ignoring the attempts to add edges that are not connected to earlier given ones (not raising any error, for simplicity). Complete the code of this class so that it achieves this behaviour. Make maximal use of the methods of Graph and do not repeat any code that can be inherited. We assume that the get_edges() method has been corrected in the way specified in Question 4.

```
class ConnectedGraph(Graph):
    # add your code below this line
```

Show the results printed in the following sequence of statements:

```
G = ConnectedGraph()

G.add_edge(1, 2)
G.add_edge(3, 4)

print(G.get_edges())  # prints:

G.add_edge(1, 3)
G.add_edge(3, 4)

print(G.get_edges())  # prints:
```