**Exam in Advanced Programming in Python (DAT515/DIT515)**

Chalmers University of Technology and University of Gothenburg
9 January 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 accepted with the grade that your lab work allows. The lab grade will be your final grade for the course, but only if you pass this exam (or a re-exam later).

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!

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

  **Value:**

  **Type:**

- `len({n % 7 for n in range(678)})`

  **Value:**

  **Type:**

- `{n % 2: n 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//n for n in range(5)}`

  **Error:**

  **Reason:**

- `{{}}`

  **Error:**

  **Reason:**

- `{'1': 'one', '2': 'two'}[2]`

  **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 whose value is a dictionary that to each stop assigns its latitude (**lat**). *Notice: the answer must be an expression, not a statement or a sequence of statements.*

**Answer:**

Assuming that the dictionary in the previous question has been given the name **latitudes**, write a Python expression whose value is the name of the southernmost tram stop (i.e. the one with the lowest latitude).

**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:

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

    def add_edge(self, a, b):
        self.adjdict[a] = self.adjdict.get(a, set())
        self.adjdict[a].add(b)

    def get_vertices(self):
        return {a for a in self.adjdict}

    def get_edges(self):
        edges = set()
        for a in self.adjdict:
            for b in self.adjdict[a]:
                if (b, a) not in edges:
                    edges.add((a, b))
        return edges
```

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.get_vertices()) # prints:

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

But watch out: the result of `get_vertices()` is not correct. This is because there is a bug in the `add_edges()` method. Show the code needed to correct this method:

**Answer:**

**Question 5** (6 p). The Graph class defined in Question 4 allows **self-loops**, that is, edges from a vertex to itself, such as (2, 2) in the above example. The following subclass is otherwise like Graph, but it forbids self-loops by simply ignoring the attempts to add them (not raising any error). 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:

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

Define a function that takes a graph of class `Graph`, possibly with self-loops, and returns a graph of class `NoSelfLoopsGraph`, by completing the following code. This function can be made very simple by using the methods of `Graph` and `NoSelfLoopsGraph`.

```
def without_self_loops(graph):
    # add your code below this line
```

Given the graph G from Question 4, write the result of

```
print(without_self_loops(G).get_edges())
```

**Answer:**