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

Chalmers University of Technology and University of Gothenburg
23 August 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 value of the following expressions? Remember that **None** is also a value!

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

  Value:

  Type:

- `len({n % n for n in range(1, 4)}) > 2`

  Value:

  Type:

- `{x: min(x, 5) for x in range(2, 8)}`

  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(4, 5)`

  Error:

  Reason:

- `{1, 2, 3} + {4, 5}`

  Error:

  Reason:

- `(lambda x: x(x))(1)`

  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 the tramnetwork dictionary, write a function that for each pair of stops A and B returns the set of lines that run through both A and B. *Hint: Notice that B can appear either after or before A in a list.*

Answer:

Using the tramnetwork dictionary, write a Python expression that returns the name of a stop that has the largest number of lines running through it. (If there are many stops with the same number, you can return any of them.)

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):
        "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 delete_edge(self, a, b):
        "delete the edge between a and b if it exists"
        self.adjdict[a] = {x in self.adjdict[a] if x != 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:

G.delete_edge(2, 3)
print(G.adjdict) # prints:
```

But watch out: there is a bug in the `delete_edge()` 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). **Weighted graphs** are graphs where each edge has a weight, an extra piece of information, such as the transition time between two tram stops in Lab 2. They can be defined as a subclass of the Graph class defined in Question 4, by adding

- an internal representation for weights assigned to each edge
- a method for setting the weight of an edge, set_weight()
- a method for getting the weight of an edge, get_weight()

Defined the class WeightedGraph by completing the code below. Make maximal use of inheritance, i.e. do not repeat code from the Graph class if not necessary. Notice that, since the graph is undirected, the weight of (a, b) is the same as the weight of (b, a). If the weight has not been set in the internal representation, or if the edge does not exist, get_weight() should return some default value. You don't need to care about the delete_edge() method.

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