Solutions to Exam in Advanced Programming in Python (DAT516/DIT515)

Chalmers University of Technology and University of Gothenburg
14 January 2025, 14 to 18, Johanneberg
Examiner: Aarne Ranta aarne@chalmers.se
Tel. 1082, mobile 0729 74 47 80

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 bring an A4 paper, hand-written on both sides, with whatever information you want to have with you.

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.

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>?
```

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].append({4})`

  Value: None

  Type: NoneType

- `[1, 2, 3] + {4}`

  Error: TypeError (AttributeError also OK)

  Explanation: Cannot concatenate list with set

- `set([1, 2, 3])`

  Value: {1, 2, 3}

  Type: set

- `{print(1//n) for n in range(1, 5)}`

  Value: {None}

  Type: set

- `'abcdefghijklmnopqrstuvwxyz'[0::100]`

  Value: 'a'

  Type: str

- `[] == {}`

  Value: False

  Type: bool

- `(lambda x, y: x < y)(2**3, 3**2)`

  Value: True

  Type: bool

- `len({0, {0}})`

  Error: TypeError (also AttributeError OK)

  Explanation: set {0} is not hashable

- `1,2**2`

  Value: (1, 4)

  Type: tuple

- `(1,2)**2`

  Error: TypeError

  Explanation: ** not defined for tuple arguments

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 line numbers (which are actually strings) that run through both "Chalmers" and "Vasaplatsen". These names are not shown in the fragment above, but you can assume that they may appear in the complete data as names of stops. *Notice: the answer must be an expression, not a statement or a sequence of statements.*

Answer:

```
 {line for line in tramnetwork['lines']
          if 'Chalmers' in tramnetwork['lines'][line] and
             'Vasaplatsen' in tramnetwork['lines'][line]}

  # even better:
 {line for line, stops in tramnetwork['lines'].items()
          if 'Chalmers' in stops and
             'Vasaplatsen' in stops}
```

Using the same definition, write a Python expression whose value is a dictionary that to each stop assigns the numbers (actually strings of digits) of the lines that the stop lies on. *Notice: the answer must be an expression, not a statement or a sequence of statements.*

Answer:

```
{stop: {line for line in tramnetwork['lines'] if stop in
        tramnetwork['lines'][line]} for stop in tramnetwork['stops']}

# even better:
{stop: {line for line, stops in tramnetwork['lines'] if stop in
        stops} for stop in tramnetwork['stops']}
```

A list of lines is also OK as the value in this dict, and even the len() of this set or list.

Grading: 5p each.

If the answer is an assignment of an expression to a variable, full points.

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)

    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 vertices(self):
        "all vertices"
        return {a for a in self.adjdict}

    def __len__(self):
        "the number of vertices"
        return len(self.vertices())
```

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)
G.add_edge(2, 1)

print(G.edges())    # prints:  {(2, 3), (1, 2), (2, 2)}
# the order in the set does not matter, but in the pairs it does
print(len(G))       # prints:  2
```

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 this method:

Answer (next page):

```
    def vertices(self):
        "all vertices"
        vs = {a for a in self.adjdict}
        for a, bs in self.adjdict.items():
            for b in bs:
                vs.add(b)
        return vs
```

The error comes out in print(len(G)), where len(G) is defined by the dunder method G.__len__(). Because of the non-redundant representation, 3 is not among the vertices, and len(G) comes out as 2 and not 3.

It was pointed out during the exam that there is also another bug in the code: if G.add_edge(2, 1) was called without also calling G.add_edge(1, 2), no edge would be added to the graph. This bug does not show in the example, but it is nonetheless a bug. One solution to that would be to rewrite

```
    def add_edge(self, a, b):
        "store new edge in the adjacency dict"
        (a, b) = sorted((a, b))
        self.adjdict[a] = self.adjdict.get(a, set())
        self.adjdict[a].add(b)
```

Fixing this bug is not required, but if you have done it, you can get points even if the len() bug remains.

Yet another solution is to change add_edge() so that it stores (a, b) in both directions. This breaks redundancy an gives only 2p. However, a correct variant is to add a statement to add_edge() that makes sure b has an entry in the adjdict, for instance

```
self.adjdict[b] = self.adjdict.get(b, set())
```

Then every vertex is represented as a key in adjdict, and vertices() comes out correct without any changes in it.
Grading:

- *print questions: 5p (3p if one is correct)*
- *bug fix: for the vertices() bug (ignoring what is written about the other issue)*
    - *5p if fixed in vertices()*
    - *5p if fixed in add_edge() with b: set()*
    - *4p if solved by fixing __len__() but not vertices()*
    - *3p for fixing add_edge() but not vertices()*
    - *2p for fixing add_edge() in a way that creates redundancy*

Question 4 (10 p). The Graph class defined in Question 3 defines undirected graphs. A **directed graph** is one where edges are like arrows, or ordered pairs: an edge from (b, a) is different from an edge (a, b). If you want edges in both directions, they have to be added separately.

Directed graphs can still be defined as a subclass of undirected graphs. Show how to do this, by making the maximal use of inheritance and hence rewriting the minimum of the code shown for Graph in Question 3. Use an adjacency dictionary as the internal representation, but remember that b now belongs to the neighbours of a if and only if there is an edge (a, b), even if b < a.

All the methods of Graph in Question 3 should be updated if necessary, but not if they can be inherited. In addition, write a method sources() which for each vertex b gives the vertices a such that there is an edge from a to b.

```
class DiGraph(Graph):

    def add_edge(self, a, b):
        "store new edge in the adjacency dict"
        self.adjdict[a] = self.adjdict.get(a, set())
        self.adjdict[a].add(b)

    def sources(self, b):
        return {a for a, bs in self.adjdict.items() if b in bs}
```

Nothing else needs to be overridden, if we assume that we have corrected vertices() in Question 3. However, if you don't assume it, it is also OK if you define it correctly here.

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

```
G = DiGraph()
G.add_edge(1, 2)
G.add_edge(2, 3)
G.add_edge(2, 2)
G.add_edge(2, 1)
```

Answer: {1: {2}, 2: {1, 2, 3}}

The order of key-value pairs or nodes in the sets does not matter.
Grading:

- *5p for correct, minimal subclassing (reductions for superfluous methods but not for a corrected version of vertices() because it was not explicit if one can assume corrected)*
- *3p for sources()*
- *2p for adjacency dictionary*

Question 5 (10 p). **Refactoring** of code means writing it in a new way that makes it clearer and, in particular, eliminates repetitions. Here is an example with three functions that are repetitive:

```
def print_small(xs):
    for x in xs:
        if x < 10:
            print(x)

def print_large(xs):
    for x in xs:
        if x >= 10:
            print(x)

def print_all(xs):
    for x in xs:
        print(x)
```

Your task is to refactor this code so that all the three functions become special cases of `print_all()`. The call

```
print_all(xs)
```

should work as before, but the other functions should come out as special cases of it:

```
print_small(xs)  becomes print_all(...)
print_large(xs)  becomes print_all(...)
```

In your answer, give the new definition `print_all()`:

```
def print_all(xs, cond=lambda x: True):
    for x in xs:
        if cond(x):
            print(x)
```

Also show how to rewrite `print_small(xs)`:

```
print_all(xs, cond=lambda x: x < 10)
```

and how to rewrite `print_large(xs)`:

```
print_all(xs, cond=lambda x: x >= 10)
```

Hint: think about how functions like `max()` and `sort()`work in Python's standard libraries and how you used `dijkstra()` in the labs.

This was meant as a "grade 5 question", a bit more tricky than the other ones.

*However, as you can see, the answers are very short. We use a default argument **cond** in the same way as the default argument **key** in max() and sort(), or the default argument **cost** in dijkstra(). This is a very compact and general solution.*

Another solution, less compact and less general, but which can give points, is to use a string default argument. For instance,

```
def print_all(xs, method='all'):
    if method='all':
      for x in xs:
          print(x)
    elif method='small':
      for x in xs:
          if x < 10:
              print(x)
    elif method='large':
      for x in xs:
          if x >= 10:
              print(x)
```

As you can see, this method does not make the code more compact, but it does solve the problem of eliminating print_small() and print_large(). Full points will, however, be given if you have a correct solution on these lines, because that is all that is specified in the question.

Yet another way to eliminate print_small() and print_large() is to rewrite

```
print_small(xs)  as print_all([x for x in xs if x < 10])
print_large(xs)  as print_all([x for x in xs if x >= 10])
```

This is not a refactoring solution, but it can give 2=1+1 points in the rewrite questions.

```
Grading:
- correct refactoring function even if not the most general one: 6p
```

- refactoring that still needs the original functions: 2p
- for the rewrites 2p each if they only call print_all() and give correct results