

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 Inspira. You can of course use separate sheets of paper to draft and experiment, but only the answers in Inspira 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/Error:

Type/Explanation:

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

Value/Error:

Type/Explanation:

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

Value/Error:

Type/Explanation:

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

Value/Error:

Type/Explanation:

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

Value/Error:

Type/Explanation:

- `[] == {}`

Value/Error:

Type/Explanation:

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

Value/Error:

Type/Explanation:

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

Value/Error:

Type/Explanation:

- `1, 2**2`

Value/Error:

Type/Explanation:

- `(1, 2)**2`

Value/Error:

Type/Explanation:

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:

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:

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:
print(len(G))       # prints:
```

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:

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):
    # add your code below this line

    def sources(self, b):
```

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:

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()`:

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

and how to rewrite `print_large(xs)`:

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