Re-exam in Advanced Programming in Python (DAT515)

Chalmers University of Technology
24 August 2023, 8:30 to 12:30, 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 1-4 of this exam to get accepted with the grade that your lab work allows.

If you have done extra labs (colouring or clustering) you will also need to get half of the points for the corresponding extra questions, in order to justify the extra points. If you have not done extra labs, your answers to those questions will not be graded.

The final result of this exam will be reported as 3, 4, 5, or rejected. As specified in the course plan, you will get

- grade 5 if you have at least 50 points from the labs, at least half of the points of the bonus questions corresponding to your labs, and at least 15 points from questions 1-4.
- grade 4 if you have at least 40 points from the labs, at least half of the points of the bonus questions corresponding to your labs, and at least 15 points from questions 1-4.
- grade 3 if you have at least 30 points from the labs and at least 15 points from questions 1-4.
- grade U otherwise

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 (12 p). Write the *value* and its *type* for each of the following expressions, or possibly an error. Remember that **None** is also a value! Examples of types are `int, float, str, bool, list, dict, set, function, NoneType.`

- `'1...5'.reverse()`
  Answer:

- `list('1...5').reverse()`
  Answer:

- `{n: list(range(n)) for n in range(1, 5)}`
  Answer:

- `len({n==n for n in range(1, 5)})`
  Answer:

- `{1, 2} == {2, 1, 2, 2, 1, 1}`
  Answer:

- `set() != {}`
  Answer:

Question 2 (4 p). Write a lambda expression for a function that takes an object and a function and applies the function twice: first to the argument object and then to the result obtained. For example, if this expression is bound to the variable **e**, then we should have

```
e(42, print) == None
e(42, lambda x: x + x) == 168
e(16, math.sqrt) == 2.0  # assuming import math
```

What are the values of the following expressions?

```
e('abc', list)
e(7, lambda x: x + 2)
```

Question 3 (6 p). Consider a dictionary of the following form, which contains numbers and their expressions in Swedish (sv) and English (en):

```
numbers = {
  1: {'sv': 'ett', 'en': 'one'},
  10: {'sv': 'tio'},
  42: {'sv': 'fyrtiotvå', 'en': 'forty-two'},
  3: {'sv': 'tre', 'en': 'three'},
  6: {'sv': 'sex', 'en': 'six'},
  20: {'sv': 'tjugo', 'en': 'twenty'},
  7: {'sv': 'sju'}
  }
```

The dictionary contains many more numbers, but not all of them, and in no specific order.

Your task is to write two Python expressions (not statements!) with the following values:

a) List of Swedish expressions for all numbers n, where 1 <= n <= 20, which are included in the dictionary but do not have an English translation there. They must be given in the numerical order. In the above dictionary, the value would be ['sju', 'tio'].

Answer:

b) List of those numbers, sorted in the numerical order, whose English and Swedish expressions have the same number of characters. For the dictionary above, the result is [1, 6, 42]. Make sure you treat numbers that are given only in one language properly!

Answer:

Question 4 (8 p).

One way to model the syntax of programming languages in compilers is to use abstract base classes for types of program elements (such as expressions and statements) and their subclasses for different kinds of elements in these types (such as addition and multiplication expressions). The base classes are **abstract** in the sense that they contain nothing:

```
class Expr:
    pass

class Stm:
    pass
```

Their subclasses contain the programming language constructs, one class for each rule in the grammar of the language. For instance, the following class defines expressions of the form `<expr> + <expr>` by combining two objects of the Expr class (that is, its subclasses) and also defining that they are shown ("printed") by showing the two expressions with a ' + ' between:

```
class AddExpr(Expr):
    def __init__(self, a, b):
        self.expr1 = a
        self.expr2 = b

    def show(self):
        return self.expr1.show() + ' + ' + self.expr2.show()
```

The following can be used for integer expressions:

```
class IntExpr(Expr):
    def __init__(self, n):
        self.expr = str(n)

    def show(self):
        return self.expr
```

Your task is to define the rest of the subclasses needed to make the following statements work:

```
ex1 = AssignStm('x', AddExpr(MulExpr(IntExpr(5), VarExpr('x')), IntExpr(3)))
print(ex1.show())
```

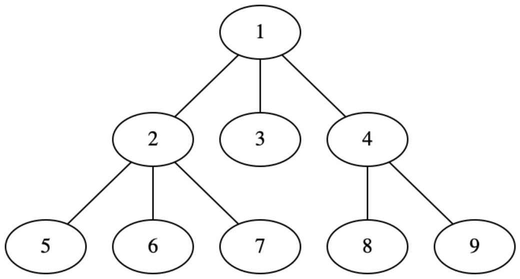
The result should be

```
x = 5 * x + 3
```

Answer to Question 4:

Extra question on graph colouring (6 p). Answer this question if and only if you have submitted the extra lab on graph colouring (either one or two parts of the lab - the question is the same in both cases).

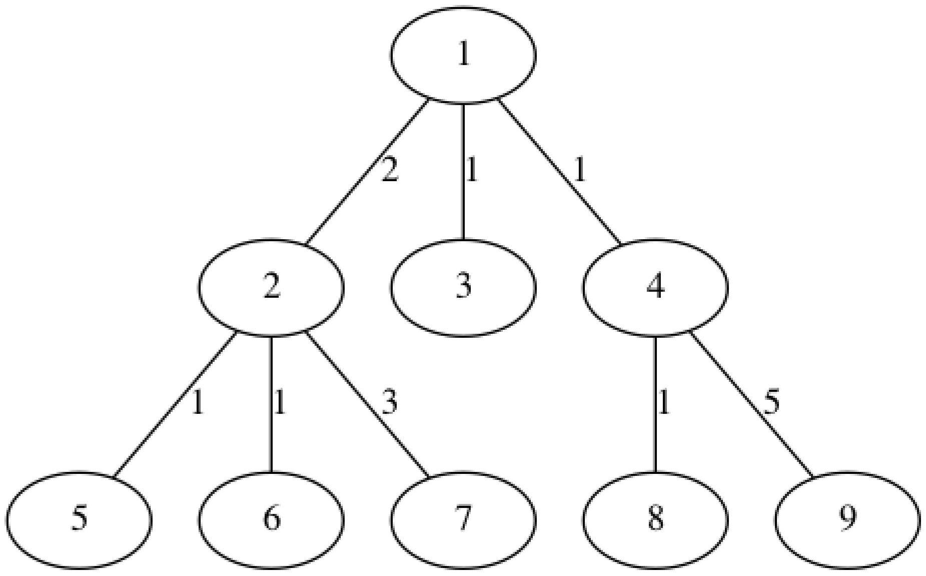
What is the minimum number of colours needed to colour the following graph with the simplify-select algorithm?



Show an order in which the vertices can be removed in the simplify phase of the simplify-select algorithm and the order in which the colours are added in the select phase. (There are many possible orders, but it is enough to show just one of them.)

Extra question on clustering (6 p). Answer this question if and only if you have submitted the extra lab on clustering.

Consider the following graph with weighted edges:



Show the k-spanning tree clusters with k = 2, 3, 4. It is enough to show which edges are removed in each case.