Tentamen för TDP002 Imperativ programmering - teoridel 2010-10-22 08-12

25 oktober 2010

Instruktioner:

- Skriv svar på varje fråga på ett separat papper.
- Uppge namn och personnummer på varje papper.
- Uppgifterna löses enskilt.
- Kursboken *Learning Python* är tillåten.
- Kommunikation med andra under tentamenstillfället är förbjudet.

Fråga:	1	2	3	4	Totalt
Poäng:	4	3	4	3	14

För betyg 3 gäller 50% rätt, för betyg 4 65% och för betyg 5 80%, avrundat till närmaste hela poäng.
Både rätt innehåll och välformulerad text krävs för full poäng på uppgifterna.
Det totala betyget på tentamen är detsamma som det lägsta betyget av teoridel och Pythondel. Det är lika många poäng på bägge delarna.
Ni får ut lösenord till tentasystemet för att göra Pythondelen av tentan efter att ni lämnat in era svar på teoridelen.

1. (4 poäng) Förklara de tre tekniska kriterierna för att utvärdera programspråk som nämns i Concepts of Programming Languages och ange åtminstone ytterligare ett kriterium som inte är relaterat till språkets tekniska uppbyggnad men som kan vara relevant att beakta när man väljer programspråk. Förklara och motivera ditt val av kriterium.

Lösningsförslag: Enligt tabell 1.1, avsnitt 1.3 i PL-boken är kriterierna Readability (läsbarhet), Writability (skrivbarhet) samt Reliability (pålitlighet). De förklaras i avsnitten 1.3.1–1.3.3. Namngivning av begreppen utan förklaring ger en poäng. En förklaring av dessa kriterier *i termer av egenskaper hos programmeringsspråk* krävs för tre poäng. Ett fjärde utvärderingskriterium som inte är relaterat till språkets tekniska specifikationer skulle kunna vara

- *antalet utvecklare som använder språket*, vilket är intressant om man vill kunna samarbeta med andra om att skriva program i språket, alternativt
- *hur utvecklat verktygsstödet är* för utveckling av program i språket, alternativt
- *om språket utvecklas på ett förutsägbart sätt*, det vill säga om det finns en implementation av språket eller flera, huruvida de skiljer sig åt och om program skrivna för en implementation går att föra över till en annan.

I allmänhet kan sägas att begreppet *kriterium* är skiljt från begreppet *egenskap*, där egenskaper är värdeneutrala medan kriterier används för att värdera (ordna) språk, det vill säga ange en preferens för ett språk jämfört med ett annat. Det i ett språk som kan sägas ha att göra med *egenskapen* ortogonalitet kan påverka *kriteriet* läsbarhet negativt eller positivt. I förklaringen av kriterier måste också *språket* skiljas från godtyckliga *program*. Läsbarhet för ett språk handlar inte om variabelnamn i program man själv skriver i språket utan om vilka språkliga konstruktioner som språket har och hur de påverkar möjligheten att skriva program som är lätta att läsa.

2. (3 poäng) Förklara vad som händer när en fil med ett program skrivet i Python exekveras, det vill säga läses in och behandlas av Pythontolken. För full poäng krävs att du hänvisar till de begrepp som angivits i kursen för respektive steg i processen.

Lösningsförslag: Kapitel 2 i Learning Python (tillåten under tentan) anger detta översiktligt och översikten ger åtminstone 1 poäng, men för full poäng krävs att ni refererar till stegen i figur 1.5 i Concepts of Programmings Languages (PL-boken), vilket vi också gick igenom under en föreläsning. Där anges att Pythontolken först delar upp filen i en sekvens meningsbärande enheter (tokens), vilket kallas lexikalisk analys. Alla kommentarer i koden undantas från bearbetning liksom övriga tecken som inte har betydelse i språket. Därefter tolkas sekvensen med tecken som utgör programmet enligt en grammatik som specificerar giltiga konstruktioner i språket. Resultatet av detta är ett abstrakt syntaxträd, vilket också kallas ett parse-träd. Trädet används sedan för att generera bytekod, vilket är det mellanliggande format som inte är beroende av vilken dator som koden exekveras på men som ändå är ett mer exekveringsanpassat format för programkod jämfört med programtext. Pythons virtuella maskin exekverar sedan programkoden när den har översatts till bytekod och läser då in från filer, interagerar med användaren och visar resultat på skärmen.

3. (4 poäng) Beskriv vad som utgör en abstrakt datatyp (ADT) genom att ange vilka *sorters* funktioner man definierar när man skapar en sådan datatyp, och ge ett eget exempel. För full poäng krävs att du anger vilka sorters funktioner som alltid måste finnas och vilka som inte alltid behöver ingå i definitionen av den abstrakt datatyp.

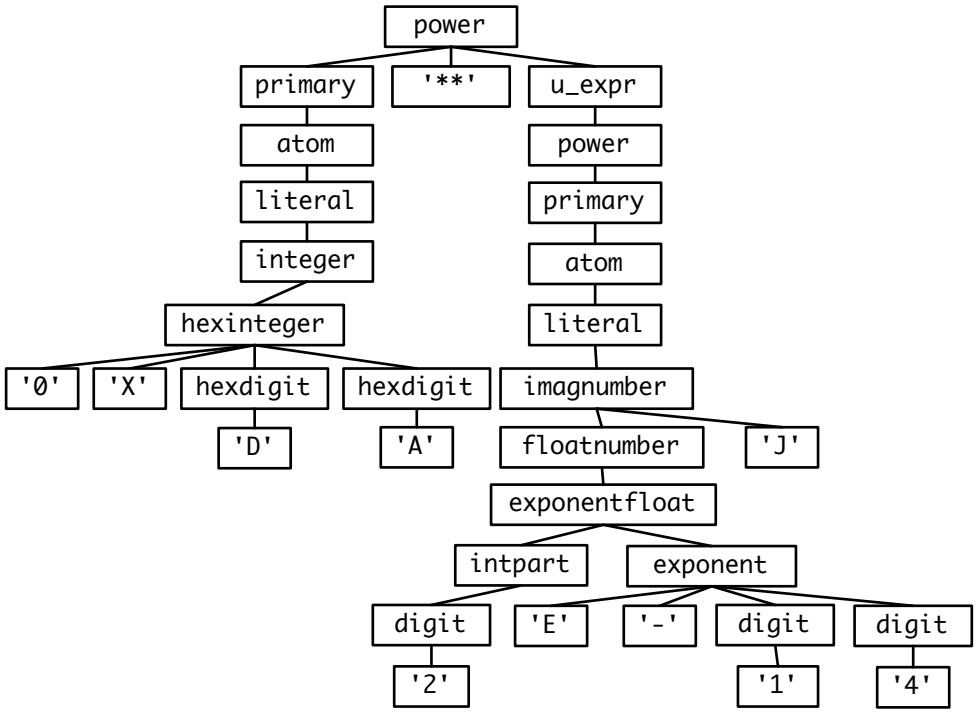
Lösningsförslag: Föreläsningsmaterialet anger fem sorters funktioner:

- *konstruktorer* som skapar ett nytt ADT-objekt,
- *selektorer* som väljer ut delar ur objekt,
- *igenkännare (alt. recognizers, predicates)* som testar om ett okänt objekt tillhör en viss ADT,
- *iteratorer* som går igenom alla delelement i objekt, samt
- *modifikatorer* som förändrar delstrukturer i objekt.

Välstrukturerade resonemang om vilka sorters funktioner som krävs och inte ger poäng. Här ges ett exempel på ett sådant resonemang.

Har man en känd, konstant uppsättning objekt av en viss typ (exempelvis spelkort) blir det inte meningsfullt att definiera en konstruktor för att kunna skapa nya, godtyckliga objekt. Är datatypen inte sådan att den består av en uppsättning mindre beståndsdelar som det är naturligt att iterera över blir inte iteratorer meningsfulla. Jämför exempelvis datatypen för enskilda spelkort som inte kan vara itererbar medan med den abstrakta datatypen för en kortlek är möjlig att iterera över.

4. (3 poäng) Använd grammatiken som bifogas för att skapa ett parse-träd av `0XDA ** 2E-14J`.



Figur 1: Parseträd för `0XDA ** 2E-14J`.

Lösningsförslag: Se figur 1. Trädet har sin rot-nod i en `power`, det vill säga uttrycket är en potens. Det är ok om ni börjat längre upp, vi visar dock det intressanta här. `0XDA` är ett hexadecimalt tal som i decimal form är `218`, men om `0` (noll) tolkats som O (versalen "O") blir ni också godkända om ni konsekvent tillämpat rätt regler för det fallet. `2E-14J` skrivs vanligen $2 \cdot 10^{-14}i$, vilket är ett imaginärt tal.

```
identifier ::=
            (letter|"_") (letter | digit | "_")*

letter ::=
            lowercase | uppercase

lowercase ::=
            "a"..."z"

uppercase ::=
            "A"..."Z"

digit ::=
            "0"..."9"

stringliteral ::=
            [stringprefix](shortstring | longstring)

stringprefix ::=
            "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"

shortstring ::=
            "'" shortstringitem* "'"
             | '"' shortstringitem* '"'

longstring ::=
            "'''" longstringitem* "'''"
               | '"""' longstringitem* '"""'

shortstringitem ::=
            shortstringchar | escapeseq

longstringitem ::=
            longstringchar | escapeseq

shortstringchar ::=
            <any source character except "\" or newline or the quote>

longstringchar ::=
            <any source character except "\">

escapeseq ::=
            "\" <any ASCII character>

longinteger ::=
            integer ("l" | "L")

integer ::=
            decimalinteger | octinteger | hexinteger

decimalinteger ::=
            nonzerodigit digit* | "0"

octinteger ::=
            "0" octdigit+

hexinteger ::=
            "0" ("x" | "X") hexdigit+

nonzerodigit ::=
            "1"..."9"

octdigit ::=
```



```
            "0"..."7"

hexdigit ::=
            digit | "a"..."f" | "A"..."F"

floatnumber ::=
            pointfloat | exponentfloat

pointfloat ::=
            [intpart] fraction | intpart "."

exponentfloat ::=
            (intpart | pointfloat)
             exponent

intpart ::=
            digit+

fraction ::=
            "." digit+

exponent ::=
            ("e" | "E") ["+" | "-"] digit+

imagnumber ::= (floatnumber | intpart) ("j" | "J")

atom ::=
            identifier | literal | enclosure

enclosure ::=
            parenth_form | list_display
               | generator_expression | dict_display
               | string_conversion | yield_atom

literal ::=
            stringliteral | integer | longinteger
               | floatnumber | imagnumber

parenth_form ::=
            "(" [expression_list] ")"

list_display ::=
            "[" [expression_list | list_comprehension] "]"

list_comprehension ::=
            expression list_for

list_for ::=
            "for" target_list "in" old_expression_list
             [list_iter]

old_expression_list ::=
            old_expression
             ["," old_expression)+ [","]]

list_iter ::=
            list_for | list_if

list_if ::=
            "if" old_expression [list_iter]

generator_expression ::=
            "(" expression genexpr_for ")"
```

```
genexpr_for ::=
             "for" target_list "in" or_test
              [genexpr_iter]

genexpr_iter ::=
             genexpr_for | genexpr_if

genexpr_if ::=
             "if" old_expression [genexpr_iter]

dict_display ::=
             "\{" [key_datum_list] "\}"

key_datum_list ::=
             key_datum ("," key_datum)* [","]

key_datum ::=
             expression ":" expression

string_conversion ::=
             "‘" expression_list "’"

yield_atom ::=
             "(" yield_expression ")"

yield_expression ::=
             "yield" [expression_list]

primary ::=
             atom | attributeref
              | subscription | slicing | call

attributeref ::=
             primary "." identifier

subscription ::=
             primary "[" expression_list "]"

slicing ::=
             simple_slicing | extended_slicing

simple_slicing ::=
             primary "[" short_slice "]"

extended_slicing ::=
             primary "[" slice_list "]"

slice_list ::=
             slice_item ("," slice_item)* [","]

slice_item ::=
             expression | proper_slice | ellipsis

proper_slice ::=
             short_slice | long_slice

short_slice ::=
             [lower_bound] ":" [upper_bound]

long_slice ::=
             short_slice ":" [stride]

lower_bound ::=
             expression
```



```
upper_bound ::=
             expression

stride ::=
             expression

ellipsis ::=
             "..."

call ::=
             primary "(" [argument_list [","]
                           | expression genexpr_for] ")"

argument_list ::=
             positional_arguments ["," keyword_arguments]
                                  ["," "*" expression]
                                  ["," "**" expression]
                | keyword_arguments ["," "*" expression]
                                    ["," "**" expression]
                | "*" expression ["," "**" expression]
                | "**" expression

positional_arguments ::=
             expression ("," expression)*

keyword_arguments ::=
             keyword_item ("," keyword_item)*

keyword_item ::=
             identifier "=" expression

power ::=
             primary ["**" u_expr]

u_expr ::=
             power | "-" u_expr
              | "+" u_expr | "\~" u_expr

m_expr ::=
             u_expr | m_expr "*" u_expr
              | m_expr "//" u_expr
              | m_expr "/" u_expr
                | m_expr "\%" u_expr

a_expr ::=
             m_expr | a_expr "+" m_expr
              | a_expr "-" m_expr

shift_expr ::=
             a_expr
              | shift_expr ( "<<" | ">>" ) a_expr

and_expr ::=
             shift_expr | and_expr "\;SPMamp;" shift_expr

xor_expr ::=
             and_expr | xor_expr "\textasciicircum" and_expr

or_expr ::=
             xor_expr | or_expr "|" xor_expr

comparison ::=
             or_expr ( comp_operator or_expr )*
```

```
comp_operator ::=
             "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
                | "is" ["not"] | ["not"] "in"

expression ::=
             conditional_expression | lambda_form

old_expression ::=
             or_test | old_lambda_form

conditional_expression ::=
             or_test ["if" or_test "else" expression]

or_test ::=
             and_test | or_test "or" and_test

and_test ::=
             not_test | and_test "and" not_test

not_test ::=
             comparison | "not" not_test

lambda_form ::=
             "lambda" [parameter_list]: expression

old_lambda_form ::=
             "lambda" [parameter_list]: old_expression

expression_list ::=
             expression ( "," expression )* [","]

simple_stmt ::= expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | pass_stmt
                | del_stmt
                | print_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | global_stmt
                | exec_stmt

expression_stmt ::=
             expression_list

assert_stmt ::=
             "assert" expression ["," expression]

assignment_stmt ::=
             (target_list "=")+
              (expression_list | yield_expression)

target_list ::=
             target ("," target)* [","]

target ::=
             identifier
                | "(" target_list ")"
```



```
                | "[" target_list "]"
                | attributeref
                | subscription
                | slicing

augmented_assignment_stmt ::=
             target augop
              (expression_list | yield_expression)

augop ::=
             "+=" | "-=" | "*=" | "/=" | "\%=" | "**="
                | ">>=" | "<<=" | "\&=" | "\textasciicircum=" | "|="

pass_stmt ::=
             "pass"

del_stmt ::=
             "del" target_list

print_stmt ::=
             "print" ([expression ("," expression)* [","]
                | ">>" expression
                  [("," expression)+ [","])

return_stmt ::=
             "return" [expression_list]

yield_stmt ::=
             yield_expression

raise_stmt ::=
             "raise" [expression ["," expression
              ["," expression]]]

break_stmt ::=
             "break"

continue_stmt ::=
             "continue"

import_stmt ::=
             "import" module ["as" name]
                ( "," module ["as" name] )*
                | "from" relative_module "import" identifier
                    ["as" name]
                  ( "," identifier ["as" name] )*
                | "from" relative_module "import" "("
                    identifier ["as" name]
                  ( "," identifier ["as" name] )* ["," ] ")"
                | "from" module "import" "*"

module ::=
             (identifier ".")* identifier

relative_module ::=
             "."* module | "."+

name ::=
             identifier

global_stmt ::=
             "global" identifier ("," identifier)*

exec_stmt ::=
```

```
             "exec" or_expr
              ["in" expression ["," expression]]

compound_stmt ::=
             if_stmt
                | while_stmt
                | for_stmt
                | try_stmt
                | with_stmt
                | funcdef
                | classdef

suite ::=
             stmt_list NEWLINE
              | NEWLINE INDENT statement+ DEDENT

statement ::=
             stmt_list NEWLINE | compound_stmt

stmt_list ::=
             simple_stmt (";" simple_stmt)* [";"]

if_stmt ::=
             "if" expression ":" suite
                ( "elif" expression ":" suite )*
                ["else" ":" suite]

while_stmt ::=
             "while" expression ":" suite
                ["else" ":" suite]

for_stmt ::=
             "for" target_list "in" expression_list
              ":" suite
                ["else" ":" suite]

try_stmt ::=  try1_stmt | try2_stmt

try1_stmt ::=
             "try" ":" suite
                ("except" [expression
                           ["," target]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]

try2_stmt ::=
             "try" ":" suite
                "finally" ":" suite

with_stmt ::=
  "with" expression ["as" target] ":" suite

funcdef ::=
             [decorators] "def" funcname "(" [parameter_list] ")"
              ":" suite

decorators ::=
             decorator+

decorator ::=
             "@" dotted_name ["(" [argument_list ","]] ")"] NEWLINE

dotted_name ::=
             identifier ("." identifier)*
```



```
parameter_list ::=
                (defparameter ",")*
               (~~"*" identifier [, "**" identifier]
                | "**" identifier
                | defparameter [","] )

defparameter ::=
             parameter ["=" expression]

sublist ::=
             parameter ("," parameter)* [","]

parameter ::=
             identifier | "(" sublist ")"

funcname ::=
             identifier

classdef ::=
             "class" classname [inheritance] ":"
              suite

inheritance ::=
             "(" [expression_list] ")"

classname ::=
             identifier

file_input ::=
             (NEWLINE | statement)*

interactive_input ::=
             [stmt_list] NEWLINE | compound_stmt NEWLINE

eval_input ::=
             expression_list NEWLINE*

input_input ::=
             expression_list NEWLINE
```