Gramática de Python 2.5

9 de marzo de 2010

La siguiente gramática corresponde a la gramática de Python 2.5[1] con algunas ligeras modificaciones.
Estructura léxica

1. Los comentarios en Python comienzan con `#` y terminan con un salto de línea.

2. Las líneas compuestas por espacios en blanco, tabuladores y comentarios se ignoran y no tienen efecto en el nivel de sangría. Lamaremos a este tipo de líneas, líneas en blanco.

3. Los espacios en blanco y tabuladores pueden separar las unidades lexicas. Estos son necesarios solo cuando si se tienen dos átomos y si ambos se encuentran concatedos de interpretan de manera diferente, por ejemplo si tenemos un `if` seguido de una `x` se puede interpretar como `ifx` si es que no estan separados por un espacio en blanco o un tabulador. Los espacios en blanco y tabuladores solo pueden aparcer dentro de literales de cadena pero no en ningún otro átomo.

4. Los saltos de línea denotados con el átomo `NEWLINE`, son significativos siempre que se encuentra fuera de un literal de cadena, excepto en las siguientes situaciones:

   - Cuando se encuentran inmediatamente después de una diagonal invertida (\), en este caso se interpreta como espacio en blanco, entre la línea actual y la siguiente.
   - Cuando se encuentran encerrados entre los siguientes tipos de paréntesis: `(` y `)`, `[` y `]`, o `{` y `}`, en este caso de nuevo los saltos de línea son tratados como espacios en blanco.

5. Pasando por alto los líneas en blanco los literales de cadena con multiples líneas (cadenas que comienzan y terminan con comillas triples), los tabuladores y espacios en blanco que se encuentran al comienzo de una línea son significativos (excepto en los casos que ya mencioncionamos anteriormente). Los espacios que se dejan de sangría en una línea, equivale al número de espacios en blanco que se dejan al inicio de esa línea.Si se encuentra un tabulador después de N espacios en blanco, este cuenta como $8-(N mod 8)$ espacios en blanco, esto es si un tabular se cuenta como el número de espacios en blanco faltantes para llegar al siguiente múltiplo de 8.

6. Si una línea tiene una sangría mayor que la línea anterior (que no sea una línea de espacios en blanco), entonces el primer átomo que se generá para dicha línea es `INDENT`. Se considera un error la primer línea (que no sea una línea de espacios en blancos) dentro de un archivo tenga sangría.

7. Si una línea tiene un sangría menor que la línea anterior (que no sea una línea de espacios en blanco), entonces se generan tantos átomos `DEDENT` al principio de dicha línea como sean necesarios para empatar con los átomos sin empatar `INDENT` presentes en las líneas previas con mayor cantidad de sangría. El final de un archivo esta precedido por tantos átomos `DEDENT` como sean necesarios para empatar todos los átomos sin empatar `INDENT`.

[1] http://www.python.org/doc/2.5/ref/grammar.txt

Los siguientes son átomos en Python.

```
keyword ::=
             "and"      | "del"      | "from"     | "not"      | "while"
           | "as"       | "elif"     | "global"   | "or"       | "with"
           | "assert"   | "else"     | "if"       | "pass"     | "yield"
           | "break"    | "except"   | "import"   | "print"
           | "class"    | "exec"     | "in"       | "raise"
           | "continue"| "finally"  | "is"       | "return"
           | "def"      | "for"      | "lambda"   | "try"

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

    (donde el lexema que empata no es una palabra reservada)

letter ::=
             lowercase | uppercase

lowercase ::=
             "a"|"b"|...|"z"

uppercase ::=
             "A"|"B"|...|"Z"

digit ::=
             "0"|"1"|...|"9"

stringliteralpiece ::=
             [stringprefix](shortstring | longstring)
             | rawstringprefix(shortrawstring | longrawstring)

stringprefix ::=
             "u" | "U"

rawstringprefix ::=
             "r" | "ur" | "R" | "UR" | "Ur" | "uR"

shortstring ::=
             "'" shortstringitem1* "'"
              | '"' shortstringitem2* '"'

rawshortstring ::=
             "'" rawshortstringitem1* "'"
              | '"' rawshortstringitem2* '"'

longstring ::=
        "'''" <cualquier sencuencia de longstringitems que no contenga "'''" sin escapar> "'''"
        | '"""' <cualquier secuencia de longstringitems que no contenga '"""' sin escapar> '"""'

rawlongstring ::=
        "'''" <cualquier secuencia de  rawlongstringitems que no contenga  "'''" sin escapar> "'''"
```

```
        | '"""' <cualquier secuencia de rawlongstringitems que no contenga '"""' sin escapar> '"""'

shortstringitem1 ::=
             <cualquier tipo de caracter excepto  "'" o salto de línea> | escapeseq

shortstringitem2 ::=
             <cualquier tipo de caracter excepto '"' o salto de línea> | escapeseq

longstringitem ::=
             <cualquier tipo de caracter excepto '\'> | escapeseq

escapeseq ::=
             '\' <cualquier caracter ASCII> | '\' <1-3 digitos octales>

rawshortstringitem1 ::=
             <cualquier tipo de caracter excepto "'" o salto de linea> | rawescapeseq

rawshortstringitem2 ::=
             <cualquier tipo de caracter excepto  '"' o salto de linea> | rawescapeseq

rawlongstringitem ::=
             <cualquier tipo de caracter excepto '\'> | rawescapeseq

rawescapeseq ::=
             '\' <cualquier caracter ASCII>

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")
```

Estructura sintáctica

Debemos considerar únicamente los programas contenidos en archivos que se generán a partir del símbolo no terminal `file_input` de la siguiente gramática de Python, nótese que no deben considerarse todas las producciones solo aquellas que esten dentro del subconjunto que se desea generar.

```
atom ::=
             identifier | literal | enclosure

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

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

stringliteral ::=
             stringliteralpiece
             | stringliteral stringliteralpiece

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 "&" shift_expr

xor_expr ::=
             and_expr | xor_expr "^" 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 ::=
            "+=" | "-=" | "*=" | "/=" | "//=" | "%=" | "**="
               | ">>=" | "<<=" | "&=" | "^=" | "|="

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
```