Simplified Python: Syntax and Control Transfer Functions

A Guide for Instructors

Contents

1 Background and Purpose

Understanding control-flow is an important Program Comprehension Task, useful for tracing and debugging programs. It is an important component of of what it means to 'know' a programming language.

This document captures the syntax and the control transfer functions in 'Simplified Python', a subset of Python expressly designed to teach concepts of control-flow to students of programming.

It is assumed that students already have 'some' knowledge of programming, not necessarily in Python. For example, they should know how to write and invoke a function that takes some arguments, know what an assignment statement does, etc.

Instructors interested in teaching the simplified version of Python and control-flow might find the notes useful. The notes provide the formal definition of the syntax and the control transfer functions. It is not necessary, and perhaps not desirable, to quote the formal definitions when elucidating the language and its control-flow structure. However, these notes could help the instructor in referring to the precise definitions, if and when necessary.

2 Syntax of Simplified Python

2.1 Main syntactic entities

The main ontological entities in Simplified Python are the following:

- Expressions
- Statements
- Blocks
- Programs
- Instructions

The syntax is expressed mostly using BNF. Additional syntactic constraints are stated in English.

- Thing ... means zero or more of Thing. (Some people use Thing* for this.)
- Thing ...? means zero or one of Thing.

2.1.1 Expressions

Expressions in Simplified Python include constants, variables or compound expressions composed with unary, or binary operators or primitive applications.

```
Exp ::= Const | Var | Exp Bop Exp | Uop Exp | Prim | (Exp)
Const ::= Num | Bool | Str
Bool ::= True | False
Bop ::= + | - | * | / | // | > | == | < | <= | >= | != | and | or
Uop ::= - | not
Prim ::= read() | int(Exp) | bool(Exp)
```

Note, expressions are *simple*: they do not contain function calls.

2.1.2 Block and Program

A block is a sequence of statements. A program is a block.

```
Blk ::= Stmt
        Stmt
  ...

Pgm ::= Blk
```

2.2 A scaffolded language hierarchy

'Simplified' Python may be seen as a family of languages. Like any family, there is a hierarchy. The simplest of the languages is the Sequential language, in which a statement is either 'pass' or an expression assignment.

2.2.1 Sequential

```
Stmt ::= pass | ExpAssign
ExpAssign ::= Var = Exp
```

2.2.2 Conditional

The Conditional language extends Sequential with the if statement:

```
Stmt ::= ... | If
If ::= if Exp:
          Blk
       else:
          Blk
```

In the language the `else` clause is mandatory, whereas in proper Python, the `else` clause is optional. Proper Python also has `elif`, which the simplified subset elides.

2.2.3 Iterative

The iterative language extends Conditional with the While, break and continue statements.

```
Stmt ::= ... | While | break | continue
While ::= while Exp:
            Blk
```

In addition, the body of a while has `continue` as the last statement.

2.2.4 Procedural

The Procedural language extends Iterative with function call definitions, call assignments and return.

```
Stmt ::= ... | Def | CallAssign | Return
Def ::=  def Id(Var, ...):
           Blk

CallAssign ::=  Var = Call
Call ::= Id(Exp, ...)
Return ::=  return Exp
```

In addition,

1. the body of a function definition must have a Return as the last statement.

2. Function Definitions can appear only in the top-level block. This restriction doesn't exist in full Python. We have this restriction to ensure 1st order functions, not higher order functions.

3. Id's are (function name) identifiers that are assumed to be non-assignable. This restriction doesn't exist in full Python.

2.3 Instructions

Python has a line-oriented syntax. Each line contains a syntact entity called an Instruction. The following grammar identifies Instructions:

```
Instr ::= pass | ExpAssign |
          if Exp: | else:
          while Exp: | break | continue |
          def id(id, ...): | CallAssign | Return
```

2.4 Program and Locations

A program of length N be seen as an array of instructions.

$$P = [0..N-1] \rightarrow Instr$$

We assume the presence of a location N, which signifies the end of the program. The set

$$L = [0..N$$

denotes the set of locations of the program P.

3 Basic Machinery for Control-Flow

For the purpose of control-flow and control transfer functions, it is fruitful to consider Programs, Blocks and Statements identified as intervals of locations. Each instruction is trivially an interval.

An interval is denoted `[f..l]`, where `f` and `l` denote the first and last index, with `f` less than or equal to `l`.

```
P=[f..l]    // Program
B=[f..l]      // Block
S=[f..l]      // Statement
I=[i..i]      // Instruction
```

```
L             // Location

first: B+S+P -> L  // + denotes union
last:  B+S+P -> L

first[f..l] = f
last[f..l] = l
```

3.1 Syntactic Constructor

```
pgm: B -> P
blk: seq[S] -> B     // sequence is nonempty
if: E, B, B -> ifS?
while: E, B -> whileS?
def(f:Id, formals:seq[Var], b: B) -> defS?
```

3.2 Statement Types

The predicates below identify syntactic subtypes.

```
whileS?: S -> Bool
ifS?: S -> Bool
defS?: S -> Bool
atomicS?: S -> Bool
breakS?: S -> Bool
contS?: S -> Bool
passS?: S -> Bool
eassignS?: S -> Bool  // ExpAssign
retS?: S -> Bool
fassignS?: S -> Bool  // Function CallAssign
top?: B -> Bool  // top-level block
```

3.3 Components

Components of syntactic entities.

```
pblk: P -> B           // The block of a program
iftest: ifS? -> ifI?
then: ifS? -> B
else: ifS? -> B
fhead: defS? -> defI?
```

```
fbody: defS? -> B
wtest: whileS? -> whileI?
wbody: whileS? -> B
stmts: B -> List[S]
called_fId: fassignS? -> Id // id of the called function
get_def: Id -> defS? // returns the function
                     // definition statement
                     // for the function id.

defname: defS? -> Id  // return the function name
                      // in a function definition

wexp: whileS? -> E  // the expression in a while.
ifexp: ifS? -> E    // the expression in a test
```

3.4 Parent

- Parent of a block is the statement enclosing it and undefined if the block is top-level.
- Parent of a statement is the enclosing block along with the index of that statement in the block.
- Parent of an instruction is the statement the instruction is part of. If the instruction is a statement on its own, then it is its own parent.

```
p: B -> S  (except the top level block)
p: S -> B x Nat
p: I -> S //
```

3.5 Closest enclosing entity

Given a statement predicate `pred?` and a statement `s` `encl(pred)(s)` returns the closest statement enclosing it that satisfies `pred?`.

The definition is written in pseudocode.

```
encl: S->bool -> [S -> S]
encl(pred?)(s) =
  let (b,_) = p(s):
    if top?(b):
        undef
    let s' = p(b):
```

```
      if pred?(s'):
        s'
      else:
         encl(pred?)(s')
```

3.6 $\text{Encl}_{\text{while}}$

Finds the closest while statement enclosing a given statement if it exists. This is used for locating the while statement enclosing a `break` or a `continue`.

```
encl_while: S -> S
encl_while = encl(whileS?)
```

3.7 Encl_{def}

Finds the closest function definition statement enclosing a given statement . Used for locating the function definition enclosing a return statement.

```
encl_def: S -> S
encl_def = encl(defS?)
```

4 Control Transfer Functions on Statements

The following definitions provide a syntactic definition of the control transfer functions.

4.1 next

```
next: S -> S

next(s:contS?)  = encl_while(s)
next(s:breakS?) = next(encl_while(s))
next(s:retS?)   = undef

next(s) =
  let (b,i) = p(s)
     if i = len(b)-1: // s is the
                      // last statement
       // in b
        if top?(b):
    undef
```

```
        else:
          next(p(b))
      else:
         stmts(b, i+1) // statement after s in b
```

4.2 true and false

```
true: S -> S
true(s:ifS?) =
    match s
  if e:
         b
     else:
    _ ::
       stmts(b, 0)

true(s:whileS?) =
    match s
   while e:
     b

        stmts(b, 0)

true(s) =
   undef

false: S -> S
false(s:ifS?) =
   match s
     if e:
    -
     else:
       b

       stmts(b,0)

false(s:whileS?) =
  next(s)

false(s) =
```

```
  undef
```

4.3 call

Returns the first statement in the body of the definition of the function being called in a function assignment statement.

```
call: S -> S
call(s:fassignS?) =
    match s
  v = f(_):
    there exists s' = def(f, _, b) and
         stmts(b, 0)
    else:
   undef
```

4.4 return

Returns all the statements that are function call assignments in which the function name is the name of the function definition enclosing the return.

```
return: S -> set[S]
return(s:retS?) =
   def f(_): _ = encl_def(s)    // find the enclosing definition f
   {next(s')| s' = _ = f(_)}    // return the nexts of all function
                                // call assignments in
        // the which f is called.
```

4.5 Statement transfer functions

The following are statement transfer functions.

```
stf? = {next, true, false, call, return}
```

5 Control Transfer Functions on Locations

The location transfer functions leverage the statement transfer functions.

```
ctf: stf? -> L -> L
ctf(stf)(i) = first(stf(p(P[i])))
err: L -> L
err(i) = last(P)
```

```
next_L = ctf(next)
true_L = ctf(true)
false_L = ctf(false)
call_L  = ctf(call)
return_L = ctf(return)
```