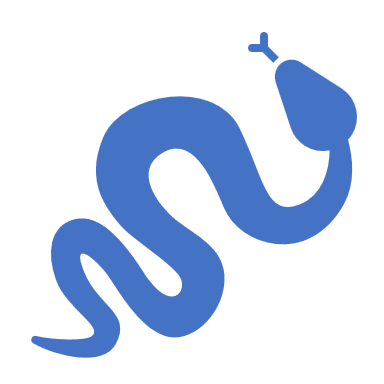
Interpreter

2018 September

<xml>

<script>



/<script>

</xml>

https://github.com/louisyang2015/interpreter

# Design

## Expression Evaluation

x = 3

x += 5

**Tokens**

|  |  |  |
| --- | --- | --- |
| Token Value | Token Value Date Type | Token Type |
| x | string | Identifier |
| = | char | Assign |
| 3 | int | Integer |

|  |  |  |
| --- | --- | --- |
| Token Value | Token Value Date Type | Token Type |
| x |  |  |
| += | string | Add Equal |
| 5 |  |  |

The tokenization needs to support simple look ahead, so that [+, =] is recognized as [+=].

**Intermediate code**

x = 3

x = x + 5

**Statements**

|  |  |
| --- | --- |
| Type | Parameters |
| Assignment | lvalue = "x"; expression = [3] |
| Assignment | lvalue = "x"; expression = ["x", +, 5] |

x = 3

x = 1 + 2 \* (3 + x) # 13

**Tokens**

|  |  |  |
| --- | --- | --- |
| Token Value | Token Value Date Type | Token Type |
| x |  |  |
| = |  |  |
| 1 |  |  |
| + | char | Add |
| 2 |  |  |
| \* | char | Multiply |
| ( | char | Left Parenthesis |
| 3 |  |  |
| + |  |  |
| x |  |  |
| ) | char | Right Parenthesis |

**Statements**

|  |  |
| --- | --- |
| Type | Parameters |
| Assignment | lvalue = "x"; expression = [2, +, 2, \*, x, \*\*, 2] |

x = 3.14

x = 1 + x # 4.14

**Tokens**

|  |  |  |
| --- | --- | --- |
| Token Value | Token Value Date Type | Token Type |
| x |  |  |
| = |  |  |
| 3.14 | double | Double |

x = 100 + -3.14e2

**Tokens**

|  |  |  |
| --- | --- | --- |
| Token Value | Token Value Date Type | Token Type |
| x |  |  |
| = |  |  |
| 100 | int | Integer |
| + | char | Add |
| -3.14e2 | double | Double |

**Tokenizing Numbers**

In general, a number looks like 3.14e-2.

The general format of a number is: digit, period, digit, 'e', sign, digit.

A state machine is used to recognize numbers:

State #: 0 1 2 3 4 5

Ending at state 0 means the token is an integer.

Ending at states 1, 2, 5 means the token is a double.

Ending at states 3 or 4 mean an invalid token.

**Expression Evaluation**

Use the Shunting-yard algorithm

### Multiline

s = """line 1

line 2"""

x = 1 + \

2

print(s,

"hello world",

x)

print(s[1:  
 3])

**Tokens**

The tokenizer needs to group these tokens on the same line.

|  |  |  |
| --- | --- | --- |
| s | = | line 1\nline 2 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | = | 1 | + | 2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| print | ( | s | , | "hello world" | , | x | ) |

### Multi-Statement

x = 1; y = 2

**Tokens**

The tokenizer needs to break semicolons into multiple lines.

|  |  |  |
| --- | --- | --- |
| x | = | 1 |

|  |  |  |
| --- | --- | --- |
| y | = | 2 |

The two token lists need to have the same indentation.

## Built-in Function

### print()

x = 2 + 3

print(x, 2\*x)

**Tokens**

|  |  |  |
| --- | --- | --- |
| Token Value | Token Value Date Type | Token Type |
| print | string | Identifier |
| ( | char | Left Parenthesis |
| x | string | Identifier |
| , | char | Comma |
| 2 | int | Integer |
| \* | char | Multiply |
| x | string | Identifier |
| ) | char | Right Parenthesis |

**Statements**

|  |  |
| --- | --- |
| Type | Parameters |
| Function Call | function\_name="print"  arguments.positional={[x], [2, \*, x]} |

print("hello", "world", sep="\*", end="...") # hello\*world...

**Statements**

|  |  |
| --- | --- |
| Type | Parameters |
| Function Call | function\_name="print"  arguments.positional={"hello", "world"}  arguments.named={"sep": "\*", "end": "..."} |

Arguments need to be a complex object that supports positional arguments and named arguments.

### Function Expansion

x = float("123.4") + int("456")  
print(x) # 579.4

In this case, the "float(...)" is a built-in function, so it's feasible to evaluate float(...) as part of the expression. But in general, the function used in expressions can be a user defined function. Single stepping through user defined function needs to be supported - meaning functions should not be resolved by Expression::eval(...).

Intermediate Code:

float("123.4")  
$temp0 = $return  
int("456")  
$temp1 = $return  
x = $temp0 + $temp1  
print(x)

**Indexer Expansion**

s2 = "abcd" + "efg" + "hijk"  
s = s2[1:3] + s2[5:7]  
print(s) # bcfg

The substring "s2[1:3]" is implemented as an "indexer(...)" function call. This is also subjected to function expansion.

Intermediate code:

s2 = "abcd" + "efg" + "hijk"  
s2[1:3]  
$temp0 = $return  
s2[5:7]  
$temp1 = $return  
s = $temp0 + $temp1  
print(s)

## Built-in Object

### String

**Assignment**

s = "hello world"

"s" is a "StringValue" object.

The string is supported by a dedicated "StringValue" class.

**Intermediate Code**

|  |  |
| --- | --- |
| Original | Intermediate Code |
| s1 = "hello world" x = 5  s2 = "je" + s1[2:x] # "jello" | s1 = "hello world"  x = 5  $temp0 = s1[2:x]  s2 = "je" + $temp0 |
| s1 = "hello world"  s2 = "je" + s1[2:6].strip() # "jello" | s1 = "hello world"  $temp0 = s1[2:6] $temp0.strip()  $temp1 = $return  s2 = "je" + $temp1 |

Due to the need to insert code in between existing code, the intermediate code should be a linked list object.

**Statements**

s1 = "x" \* 3

|  |  |
| --- | --- |
| Type | Parameters |
| Assignment | lvalue = "s1"  expression = ["x", \*, 3] |

s3 = "abc" + " efg"

|  |  |
| --- | --- |
| Type | Parameters |
| Assignment | lvalue = "s3"  expression = ["abc", +, " efg"] |

b = 'e' **in** s3 # True

|  |  |
| --- | --- |
| Type | Parameters |
| Assignment | lvalue = "b"  expression = ["e", in, s3] |

b = s3.isdigit() # False

|  |  |
| --- | --- |
| Type | Parameters |
| Function Call | object = s3  function name = "isdigit"  arguments = {} |
| Assignment | lvalue = "s3"  expression = [$return] |

**Slice notation**

s2 = "abc"[1:2]

|  |  |
| --- | --- |
| Type | Parameters |
| Assign | lvalue = "s2"  expression =  {  SliceExpression  {  value = StringValue("abc")  start expression = 1  stop expression = 2  step expression = null  } } |

When the "SliceExpression" is evaluated, it will return a substring. This is a bit memory inefficient, but easy to implement.

s2 = "abcd" + "efg" + "hijk"  
  
s = s2[2:5] + s2[3:9:2]

|  |  |
| --- | --- |
| Type | Parameters |
| Assign | lvalue = "s"  expression = {  SliceExpression  {  var\_name = "s2"  start expression = 2  stop expression = 5  step expression = null  },  Add,  SliceExpression  {  var\_name = "s2"  start expression = 3  stop expression = 9  step expression = 2  } } |

## Flow Control

### If

**Basic "if" statement**

if x==0:

print(0)

**Statements**

|  |  |  |
| --- | --- | --- |
| Address | Type | Parameters |
| 0 | Conditional | expression = ["x", ==, 0]; true\_addr=current+1; false\_addr = end\_if |
| 1 |  | Function call to "print" |
| end\_if = 2 |  |  |

**Address generation procedure - the label ID system**

Initially, upon seeing the very first line of the "if" statement, put in a fake negative address:

|  |  |
| --- | --- |
| Program Statements (after processing if == 0) | |
| Address | Statement |
| 0 | Conditional statement, **false\_addr = -1** |

Valid addresses are >= 0, so the -1 is actually an "label id".

In this particular example, the "end\_if" label has an ID of 1. Until the address is truly known, the "false\_addr" takes a value of -1.

The true address should be stored inside an array.

|  |  |
| --- | --- |
| Program Addresses (upon allocating label ID 1) | |
| Index (the label ID) | Address |
| 0 | Not Used |
| **1** | -1 |

At the very beginning of the "if" statement, the true address of label ID = 1 is unknown. So it's -1.

After processing the print statement, the program statements look like:

|  |  |
| --- | --- |
| Program Statements (after processing print(0)) | |
| Address | Statement |
| 0 | Conditional statement, false\_addr = -1 |
| 1 | Function call to "print" |

At this point, the label "end\_if", aka label ID = 1, is resolved as 2 (the next address).

|  |  |
| --- | --- |
| Program Addresses (after processing the print(0)) | |
| Index (the label ID) | Address |
| 0 | Not Used |
| **1** | 2 |

After all of the program statements have been constructed, go back and resolve all the negative addresses. A address of -1 means to look at index 1 of the address array for the true address.

|  |  |
| --- | --- |
| Program Statements (after address resolution) | |
| Address | Statement |
| 0 | Conditional statement, **false\_addr = 2** |
| 1 | Function call to "print" |

**Full "if" statement**

x = 3

if x==0:

print(0)

elif x==1:

print(1)

elif x==2 or x==3:

print("2 or 3")

else:

print("not 0 and not 1")

**Statements**

|  |  |  |
| --- | --- | --- |
| Address | Type | Parameters |
| start\_if[0] | Conditional | expression = ["x", ==, 0]; true\_addr=current+1; false\_addr = start\_if[1] |
|  |  |  |
|  | Jump | addr = end\_if |
| start\_if[1] | Conditional | expression = ["x", ==, 1]; true\_addr=current+1; false\_addr = start\_if[2] |
|  |  |  |
|  | Jump | addr = end\_if |
| start\_if[2] | Conditional | expression = ["x", ==, 2, or, "x", "==", 3]; true\_addr=current+1; false\_addr = else |
|  |  |  |
|  | Jump | addr = end\_if |
| else |  |  |
|  |  |  |
| end\_if |  |  |

It's quite possible for the "else" block to be empty - for it to not exist at all.

The blue boxes, "start\_if[0]", "start\_if[1]", ... "end\_if", are all "label IDs". These are stored as negative address values in the conditional and jump statements.

**Single line "if" statements:**

if x == 3: print("x is 3")

The intermediate code needs to "standardize" it into the two line form:

if x == 3:   
 print("x is 3")

### While

|  |  |
| --- | --- |
| 0  1  2  3  4  5  6  7 | i = 0  while i < 10:  i += 1  if i==3:   continue  print(i)  if i == 7:   break |

**Statements**

|  |  |  |
| --- | --- | --- |
| Address | Type | Parameters |
| start\_while | Conditional | expression=[i, <, 10]; true\_addr=current+1; false\_addr=end\_while |
|  |  |  |
|  | Jump | addr=start\_while |
| end\_while | End While |  |

**Supporting "break" and "continue"**

The "break" statement jumps to the end of the loop.

The "continue" statement jumps to the beginning of the loop.

The statements are generated recursively. In the above example:

construct\_statements(0,7)  
 |🡪 construct\_statements(2,7)  
 |🡪 construct\_statements(4,4)  
 |  
 |🡪 construct\_statements(7,7)

Each construct\_statements(...) needs access to the current (aka innermost) "while" or "for" label IDs - so that the "break" and "continue" statements will have the correct jump targets.

**Intermediate code to supporting function un-nesting**

To support function un-nesting, the while loop's intermediate code needs to allow for the start of the while loop to be a different place than the conditional of the while loop.

Example:

i = 0  
while s[i] != 'w': i += 1

Intermediate code:

i = 0  
while  
s [ i ]  
$temp0 = $return  
while $temp0 != "w" :  
    i = i + 1

The while loop starts after the first "while", at line "s[i]". The conditional is later, at the second while line. Allowing the conditional to be on a different line than the "while\_start" part of the "while" loop is the key to simplifying complex while statements.

The intermediate code generation of the "while" loop goes through several stages.

Original code:

while s[i] != 'w': i += 1

First add the standalone "while" marker:

while  
while s[i] != 'w': i += 1

Next break any single line "while" into two lines:

while  
while s[i] != 'w':   
 i += 1

Apply standard function un-nesting:

while  
s[i]  
$temp0 = $return  
while $temp0 != 'w':   
 i += 1

### For

x = 11

for i in range(1, 17, 2):

if i==5: continue

print(i)

if i==x: break

**Intermediate Code**

$temp0 = range(1, 17, 2)

for i in temp0:

    if i==5:

        continue

    print(i)

    if i==x:

        break

**Indentation aware temporary variable**

In the example above, the inner code must not use "$temp0". The "$temp0" needs to remain "range(1, 17, 2)" until the loop is done.

The issue is that the temporary variables are cleared, aka reused, after each statement is generated.

The temporary variable system needs to be indentation aware. The $temp0 is created at indentation 0. While generating the intermediate code for the body of the "for" loop, the indentation is 4. So temporary variables generated at indentation 4 is available for reuse by the indentation 4 statements, but the temporary variable at indentation 0, the "$temp0", needs to be preserved until the indentation drops back to 0.

**Statements**

The "for" loop in general uses an iterator object. So the run time has to support the range object.

|  |  |  |
| --- | --- | --- |
| Address | Type | Parameters |
|  | Assignment | lvalue="$temp0"; expression=[range, (, 1, 17, 2, )] |
|  | Functional Call | object\_name = $temp0, function\_name = reset |
| start\_for | Functional Call | object\_name = $temp0, function\_name = has\_more\_elements |
|  | Conditional | expression = $return; true\_addr=current+1; false\_addr=end\_for |
|  | Function Call | obj\_name = $temp0, function\_name = next\_element |
|  | Assignment | lvalue="i"; expression = $return |
|  |  |  |
|  |  |  |
|  | Jump | addr=start\_for |
| end\_for |  |  |

The iteration interface:

reset() - to reset the iterator to point to the very first element

has\_next\_element() - to decide whether to continue the for loop, or to exit the for loop

next\_element() - to retrieve an element from the iterator

The "break" statement jumps to the end of the loop (label ID end\_for).

The "continue" statement jumps to the beginning of the loop (label ID start\_for).

## Function

**Calling side statements**

**def** f2(x1, x2):  
 **return** x1 + x2  
  
x = f2(1, 2)

Intermediate code:

f2(1, 2)  
x = $return

|  |  |
| --- | --- |
| Type | Parameters |
| Function Call | function\_name=f2; arguments={[1,+,2], [3,+,4]} |
| Assignment | lvalue="x"; expression=$return |

**Nested function expansion**

Nested function calls need to be expanded first.

f2(f2(1+2, 3+4),   
 f2(5+6, f(7+8)))

Intermediate code:

f(7+8)  
$temp2 = $return  
f2(5+6, $temp2)

$temp1 = $return  
f2(1+2, 3+4)

$temp0 = $return

f2($temp0, $temp1)

**Function side basic case**

**def** f(x, y):  
 print(x, y)  
  
f(x=1, y=2)  
f(y=1, x=2)

|  |  |  |
| --- | --- | --- |
| Address | Type | Parameters |
|  | Jump | Addr = end\_func |
| start\_func | Function Arg Assign | lvalue="x", position=0,  default=null |
|  | Function Arg Assign | lvalue="y", position=1,  default=null |
|  |  |  |
|  | Return | return\_value = None |
| end\_func |  |  |

Normal code needs to skip over the function definition - so the very first instruction is to skip over the function definition.

The "FunctionArguments" (C#) class puts arguments into:

List<Expression> positional\_args

Dictionary<string, Expression> named\_args

The "Function Arg Assignment" will have to look at the "positional\_args" first, and if nothing is found there, look at "named\_args".

The assignments need to happen in the symbol table local to the function.

The "default=null" is different from "default=None". There is really a Python value call "None" that can be the default argument.

**Default arguments**

**def** f(x, y=1, z=2):  
 print(x, y, z)  
  
f(1, z=3)  
f(x=3)

|  |  |  |
| --- | --- | --- |
| Address | Type | Parameters |
|  | Jump | Addr = end\_func |
| start\_func | Function Arg Assign | lvalue="y", position=1,  default=1 |

**Global Variable**

x\_global = 5  
  
**def** change\_x\_global(new\_x\_global):  
 **global** x\_global  
 x\_global = new\_x\_global  
  
change\_x\_global(10)  
print(x\_global)

|  |  |
| --- | --- |
| Type | Parameters |
| Global | var\_name = "x\_global" |

The "symbol table" object internally needs to have multiple lookup tables:

class SymbolTable

global  
symbols

local symbol table

global symbol table

local symbol table

global  
symbols

global  
symbols

local symbol table

Function Call Stack

**\*args and \*\*kwargs support**

**def** f(x, \*args, \*\*kwargs):  
 print("x =", x)  
  
 print("\*args:")  
 **for** arg **in** args:  
 print(arg)  
  
 print("\*\*kwargs:")  
 **for** key **in** kwargs.keys():  
 print(kwargs[key])  
  
  
f(1, 2, 3, "abcd", "efg", z=456)  
  
f(x=123, y=234, z=456)

**\*args** - The "args" is the list of positional arguments beyond the stated arguments. In the above example, the function states one positional argument, "x". So args = positional\_args starting at index = 1.

**\*\*kwargs** - This is trickier than "args". The "x=123" belongs to "x", not to "kwargs", so the "kwargs" is not simply all of the named arguments. It has to be the named arguments are have not been "used". In this example's "f(x, \*args, \*\*kwargs), extracting the "x" uses up the "x", so that it doesn't "count" the "kwargs" dictionary.

## List

**List creation**

y = 10  
x = [1 + y, 2\*y, 3 + 2\*y, 4, "hello"]  
print(x)

Currently there is Expression::eval(...) 🡪 Value

Mirroring this situation we need: ListExpression::eval(...) 🡪 ListValue

ListValue - C# implementation of the Python List  
ListExpression - A list of expressions. Call "eval(...)" of each expression to get a list of values.

The assign statement needs to support a list of expressions. More specifically, an expression can contain a list of expressions. When evaluated, the list of expressions results in a list of values.

Assign statement   
{  
 lvalue = "x",  
 expression =   
 {  
 list expression   
 {  
 {1 + y},  
 {2 \* y},  
 {3 + 2 \* y},  
 {4}  
 {"hello"}  
 }  
 }  
}

**Operator support**

x = [1, 2, 3, 4]  
print(2 **in** x)  
print(-1 **not in** x)

y = [10, 11, 12]  
print(x + y)  
print(x \* 3)  
print(3 \* x)

**slice notation support**

# slice  
x = [1, 2, 3, 4]  
  
print(x[0])  
x[0] = 10  
print(x)  
  
print(x[1:3])  
x[1:3] = [7, 8, 9, 10]  
print(x)  
  
print(x[1:5:2])  
x[1:5:2] = [0, 1]  
print(x)  
  
y = 1  
x[y:y+4:2] = [20, 20]  
print(x)  
  
y = x[:4]  
y[0] = 30  
print(x)  
print(y)  
  
x[0] = 1  
print(y[x[0]])

The slice operator mean different things depend on how it's used.

In one case, the slice operator is assignment into a list.

In another case, the slice operator extracts a subset of the list.

When used in a for loop, the slice operator generates an iterator:

**for** i **in** x[1:3]:  
 print(i, end=' ')

When used with a "del" keyword, the slice operator deletes part of a list:

**del** x[0:6:2]

To support the slice feature, lazy evaluation is needed. The "x[0:6:2]" is stored as a "SliceExpression", which is a wrapper around the "x".

* For the assign statement, the "SliceExpression" will extract a new list.
* For the for loop and print(), the "SliceExpression" is an iterator over the inner value.
* For the "del" keyword, the "SliceExpression" just provides the start, stop, and step numbers.

print(x[1:5:2])

Statement:

|  |  |
| --- | --- |
| Type | Parameters |
| Function Call | function name = "print"  arguments  {  expression =  {  SliceExpression  {  var\_name = "x"  start expression = 1  stop expression = 5  step expression = 2  }  } } |

x[y:y+4:2] = [20, 20]

Statement:

|  |  |
| --- | --- |
| Type | Parameters |
| Slice Assign | SliceExpression lvalue =  {  var\_name = "x"  start expression = y  stop expression = [y + 4]  step expression = 2 }  Expression expression =  {  ListExpression = {20, 20}  } |

**del** x[0:6:2]

Statement:

|  |  |
| --- | --- |
| Type | Parameters |
| Keyword | keyword = "del"  expression =  {  SliceExpression  {  var\_name = "x"  start expression = 0  stop expression = 6  step expression = 2  }  } |

**Special default case**

In

print(x[::-1])

The "end" default is not simply -1 --- because x[:-1:-1] is different from x[::-1]. The index effectively does become -1 though. These two scenarios need to be handled separately:

* user enters -1
* user enters nothing, as in x[::-1].

**support for deleting elements - via slice notation, del keyword, and the .clear() method**

x = [1, 2, 3, 4] \* 3  
x[2:5] = []  
  
  
x = [1, 2, 3, 4]  
x \*= 3  
**del** x[3]  
  
y = 2  
**del** x[y:y+4:2]

x.clear()  
  
**del** x

**List methods: .append(), .copy, .extend(), .insert(), .pop(), .remove(), .index(), .count(), .sort()**

# .append()  
x = [1, 2, 3, 4]  
  
x.append([6, 7]) # creates nested list  
  
# .copy()  
y = x.copy()  
  
y[0:4:2] = [0, 0]  
y[5][0] = 106 # index 0, 2 are not affected; index 5 affected  
y[5][1] = 107  
  
# .extend() .insert() .pop() .remove()  
x = [1, 2, 3, 4]  
x.extend([5, 6, 7])  
  
x.insert(4, 20)  
x.insert(4, [100, 101, 102]) # nested list  
  
i = x.pop(4)  
  
i = x.pop()  
  
x.remove(20)  
  
x.reverse()

# .index() .count()  
x = [1, 2, 2, 2, 3, 4]  
print(x.index(2))  
print(x.index(2, 3))  
# print(x.index(10)) # ValueError not supported  
# print(x.index(3, 0, 3)) # ValueError not supported  
print(x.count(2))  
print(x.count(10))  
  
# .sort()  
x = [3, 1, 2, 6, 5]  
x.sort()  
  
x.sort(reverse=**True**)

**Slice notation method calls**

print(x[1:].index(2, 3))

Intermediate code:

$temp0 = x[1:]  
print($temp0.index(2,3))

which then becomes:

$temp0 = x[1:]  
$temp0.index(2,3)  
$temp1 = $return  
print($temp1)

**Built-in functions support for list**

len(x)  
min(x)  
max(x)

r = list(range(0, 10, 2))

**For loop support**

To use the for loop on a list directly, the list needs to implement the iteration methods: next\_element(), has\_next\_element(), and reset().

# for loop support  
x = [1, 2, 3, 4, 5, 6]  
**for** i **in** x:  
 print(i, end=' ')  
print()  
  
x = ['a', 'b', 'c', 'd', 'e', 'f']  
**for** i **in** range(0, len(x)):  
 print(x[i], end=' ')  
print()  
  
**for** i **in** range(0, len(x), 2):  
 print(x[i], end=' ')  
print()

Intermediate code expansion needs to be done for looping over slice:

**for** i **in** x[1:3]:  
 print(i, end=' ')  
print() # b c

Intermediate code:

$temp0 = x[1:3]  
for i in $temp0:  
 print(i, end = ' ')  
print()

This assignment to "$temp0" should not duplicate the list, to save memory.

However, this is a bit different from Python's behavior.

The Python code:

x = [5, 6, 7, 8, 9, 10, 11]  
**for** i **in** x[1:5]:  
 x[i - 5 + 1] += 10  
 print(i, end = ' ')  
print() # 6 7 8 9

does modify the list "x", but the values printed out are the old values. This means a copy is made before (or during) the for loop.

The guess is that Python has copy on write - it doesn't normally make a copy to save memory, but seeing the "x[i-5+1] += 10" line, Python makes a copy.

Essentially this interpreter is taking a simpler route - no implementation of copy on write. Have the user manually make a copy, as in:

x = [1, 2, 3, 4]  
x\_copy = x[1:]  
**for** i **in** x\_copy:  
 x[1] = 10  
 x[2] = 10  
 x[3] = 10  
 print(i, end = ' ')  
  
print() # 2 3 4  
print(x) # [1, 10, 10, 10]

There are two kinds of behavior:

x\_copy = x[1:]  
$temp0 = x[1:]

The first makes a copy while the second doesn't. This is implemented by looking at the variable name. "x\_copy" is a user variable, so a copy is made. The "$temp0" is a temporary variable, so no copy is made.

**String methods that require list support:**

" ".join(["a", "b", "c"])

"ab cde".partition(' ')

"ab cde".rpartition(' ')

.split(), .rsplit(), .splitlines()

### Slice nesting

temp = x[y[1+z]]

The start expression for "x" is a slice expression.

|  |  |
| --- | --- |
| Type | Parameters |
| Assign | lvalue = "temp"  expression =  {  SliceExpression  {  var\_name = "x"  start expression =  {  SliceExpression  {  var\_name = "y"  start expression = {1 + z}  }}} } |

temp = x[1][2]

The inner slice expression is x[1]. The outer slice expression is (x[1])[2].

|  |  |
| --- | --- |
| Type | Parameters |
| Assign | lvalue = "temp"  expression =  {  SliceExpression  {  value = SliceExpression   {  var\_name = "x"  start expression = 1  }  start expression = [2]  } } |

To support nesting, a slice should be implemented as a kind of list. The difference between a slice and a list is that the list index and list length are being recomputed using the "start, stop, step" values. So a slice should be a special wrapper around the list. This helps to support nesting because a list, a slice(list), a slice(slice(list)) all look like a list and can all be manipulated with the same code.

**Sequentially Nested Slice Implementation (for List<int>)**

The following code is the C# implementation for:

list1 = list(range(0, 10, 1))  
print(list1)  
print(list1[2:10:1])  
print(list1[2:10:1][0:6:2])

using System;

using System.Collections;

using System.Collections.Generic;

class Program

{

  static void Main(string[] args)

  {

    IList<int> list = new List<int>();

    for (int i = 0; i < 10; i++)

      list.Add(i);

    var slice = new Slice(list, 2, 10, 1);

    var slice2 = new Slice(slice, 0, 6, 2);

    print(list);

    print(slice);

    print(slice2);

    Console.Read();

  }

  static void print(IList<int> list)

  {

    foreach (var i in list)

      Console.Write(i + " ");

    Console.WriteLine();

  }

}

class Slice : IList<int>

{

  IList<int> list;

  int start, stop, step;

  public Slice(IList<int> list, int start, int stop, int step)

  {

    this.list = list;

    this.start = start;

    this.stop = stop;

    this.step = step;

  }

  /// <summary>

  /// Translate between the slice list's "index" and the "real\_index"

  /// that is used to access the underlying "list".

  /// </summary>

  int real\_index(int index)

  {

    int real\_index = start + index \* step;

    // check real\_index versus "stop"

    if (step > 0)

    {

      if (real\_index >= stop)

        throw new Exception("Index out of bound");

    }

    else if (step < 0)

    {

      if (real\_index <= stop)

        throw new Exception("Index out of bound");

    }

    return real\_index;

  }

  public int this[int index]

  {

    get { return list[real\_index(index)]; }

    set

    {

      list[real\_index(index)] = value;

    }

  }

  public int Count

  {

    get

    {

      int num\_elements = 0;

      if (step > 0)

        num\_elements = (int)Math.Floor((double)(stop - 1 - start) / step) + 1;

      // for positive step, the final value is (stop - 1)

      // (stop - 1 - start) is number of elements in addition to the starting value

      // +1 at the very end to include the starting element

      else if (step < 0)

        num\_elements = (int)Math.Floor((double)(stop + 1 - start) / step) + 1;

      // for negative step, the final value is (stop + 1)

      if (num\_elements < 0) num\_elements = 0;

      return num\_elements;

    }

  }

  public bool IsReadOnly { get { return false; } }

  public void Add(int item)

  {

    // Using Python's .append() on a slice seems to do nothing

    // to the underlying list, and the .append() itself

    // returns "None".

    throw new Exception("Calling append() on a slice is not supported.");

  }

  public void Clear()

  {

    // Using Python's .clear() on a slice seems to do nothing

    // to the underlying list.

    throw new Exception("Calling clear() on a slice is not supported.");

  }

  public bool Contains(int item)

  {

    int count = Count; // prevent re-computation during for loop below

    for (int i = 0; i < count; i++)

      if (list[real\_index(i)] == item) return true;

    return false;

  }

  public void CopyTo(int[] array, int arrayIndex)

  {

    int count = Count; // prevent re-computation during for loop below

    for (int i = 0; i < count; i++)

      array[arrayIndex] = list[real\_index(i)];

  }

  public IEnumerator<int> GetEnumerator()

  {

    return new SliceEnumerator(this);

  }

  public int IndexOf(int item)

  {

    int count = Count; // prevent re-computation during for loop below

    for (int i = 0; i < count; i++)

      if (list[real\_index(i)] == item) return i;

    return -1;

  }

  public void Insert(int index, int item)

  {

    throw new NotImplementedException();

  }

  public bool Remove(int item)

  {

    // Using Python's .remove() on a slice seems to do nothing

    // to the underlying list.

    throw new Exception("Calling remove() on a slice is not supported.");

  }

  public void RemoveAt(int index)

  {

    throw new NotImplementedException();

  }

  IEnumerator IEnumerable.GetEnumerator()

  {

    throw new NotImplementedException();

  }

}

class SliceEnumerator : IEnumerator<int>

{

  Slice slice;

  int current\_index = -1;

  public SliceEnumerator(Slice slice)

  {

    this.slice = slice;

  }

  public void Reset()

  {

    current\_index = -1;

  }

  public bool MoveNext()

  {

    if (current\_index < slice.Count - 1)

    {

      current\_index++;

      return true;

    }

    else

      return false;

  }

  public int Current

  {

    get { return slice[current\_index]; }

  }

  object IEnumerator.Current

  {

    get { return slice[current\_index]; }

  }

  public void Dispose() { }

}

### Dictionary

**DictionaryValue, ExpressionDictionary**

The dictionary implementation mirrors the list implementation, with a "value" and "expression" class.

DictionaryValue - C# implementation of the python dictionary  
ExpressionDictionary::eval(...) 🡪 ValueDictionary

**Hash function**

For a dictionary to work, the dictionary key's hash function must be properly implemented.

Two class instances that are equal need to produce the same hash value. This cannot be defaulted to the "object" class's "GetHashCode()". Only the programmer knows which fields inside of a class contribute to the equality of two class instances.

The "GetHashCode()" has been overwritten for "DynamicValue" and "StringValue". So only numbers and strings can be used for dictionary keys. In both cases, the "GetHashCode()" is just:

return value.GetHashCode();

# Implementation

## Tokenizer

**File: Tokenizer.cs**

**Organization**

Each line of the source code corresponds to a "TokenList", which then in turn contain individual "Token" objects.

class TokenList

int line\_number

Token

Token

Token

The "line\_number" points to the location in the original source code.

### Token

Each Token has a "type" and a "value":

class Token

{

  public readonly TokenType type;

  public readonly object value;

There are many token types:

enum TokenType

{

  Add, Add\_Equal, And, As, Assert, Assign, Break,

  Class, Colon, Comma, Continue, Def, Del, Divide,  
 ...

**Constant Tokens**

Most of these token types exist in only one version. For example, the "Add" token is always:

type = TokenType.Add  
value = '+'

To save memory (slightly), tokens that take only one form is in

class Token

class Constants  
 static public readonly Token Add, Add\_Equal, And, As, Assert,   
 ...

All the "Add" tokens in this program is really "Token.Constants.Add".

### Mapping source code to tokens

The mapping from string values to tokens are established in "Token" static constructor:

class Token  
{

static Token()

{

// Initialize static data structures

// Initialize symbol\_mappings

symbol\_mappings = new Dictionary<char, Token>();  
 symbol\_mappings.Add('+', Constants.Add);

symbol\_mappings.Add('=', Constants.Assign);

symbol\_mappings.Add('&', Constants.Bitwise\_And);  
 ...  
 // Initialize keyword\_mappings  
 keyword\_mappings = new Dictionary<string, Token>();

keyword\_mappings.Add("and", Constants.And);

keyword\_mappings.Add("as", Constants.As);

keyword\_mappings.Add("assert", Constants.Assert);  
 ...

The "Token" class has static methods that use these mappings to look for symbols and keywords inside strings.

### Call Tree

class Tokenizer  
 Tokenizer(string[] source)  
 |🡪 TokenList get\_token\_list(ref int line\_number, ref int char\_number)

|🡪 Token get\_string\_token(ref int line\_number, ref int char\_number)  
 |  
 |🡪 Token get\_token(ref int line\_number, ref int char\_number)  
 |  
 |  
 |🡪 class Token

(Token, int) parse\_symbol(char current\_char, ...)

Token parse\_keyword(string token)

(Token, int) parse\_number(string substring, int line\_number)

Token parse\_identifier(string token, int line\_number)

Token create\_string\_token(string token)

bool is\_token\_end(char c)

**Counting parenthesis**

Counting the balance of "( )", "[ ]", and "{ }" is part of checking for syntax error.

TokenList :: int count\_braces(bool excessive\_left\_token\_error)

|🡪 int count\_token\_balance(Token left, Token right,...)

TokenList :: int count\_brackets(bool excessive\_left\_token\_error)

|🡪 int count\_token\_balance(Token left, Token right,...)

TokenList :: int count\_parenthesis(bool excessive\_left\_token\_error)  
 |🡪 int count\_token\_balance(Token left, Token right,...)

Having more right token than left token, such as having more ']' than '[', always cause an exception to be thrown.

However, having more left token might be okay, depending on the situation. So the "excessive\_left\_token\_error" flag controls whether having excessive left token is considered an error.

**Skipping spaces and lines**

All space skipping and line skipping decisions happen inside "get\_token\_list(...)"

Tokenizer(string[] source)  
 |🡪 TokenList get\_token\_list(ref int line\_number, ref int char\_number)

|🡪 Token get\_string\_token(ref int line\_number, ref int char\_number)  
 |🡪 Token get\_token(ref int line\_number, ref int char\_number)

The "Tokenizer()" constructor doesn't change the "line\_number" and "char\_number" at all.

The "get\_string\_token(...)" and "get\_token(...)" only changes "line\_number" and "char\_number" from the start of the token to the character right after the end of the token.

Skipping the spaces in between tokens, as well as moving to new lines, happen completely inside "get\_token\_list(...)".

"get\_token\_list(...)", with just the space skipping and line skipping parts, are shown below.

Color code: space skipping; line skipping

TokenList get\_token\_list(ref int line\_number, ref int char\_number)

{

int indentation = find\_non\_whitespace(line\_number, char\_number);

while (true)

{

// get next token

char\_number = find\_non\_whitespace(line\_number, char\_number);

var token = get\_token(ref line\_number, ref char\_number);

if (token != null)

{

   // handle ';' line termination

  if (token.type == TokenType.Semicolon)

return token\_list;

}

else

{

   // token == null case

   // End of the line, but look for line continuations.

   line\_number++;

  char\_number = 0;

   if (line\_number >= source.Length) return token\_list;

  if (token\_list[-1].type == TokenType.Slash)

   {

      // handle '\' line continuation

     token\_list.remove\_last(1);

   }

  else if (token\_list.count\_parenthesis(excessive\_left\_token\_error: false) > 0

     || token\_list.count\_braces(excessive\_left\_token\_error: false) > 0

     || token\_list.count\_brackets(excessive\_left\_token\_error: false) > 0)

  {

     // This means '(', '[', or '{' line continuation.

     // Don't return, keep going.

  }

   else

     return token\_list;

}

There is only one location for space skipping and one location for line skipping.

If (line\_number, char\_number) has reached the end of the line, the "get\_token(...)" will return a null, then the "get\_token\_list(...)" will do the "line\_number++".

After skipping a line, the "get\_token\_list(...)" will see if the newline is a new statement, or if it's continuation of the same statement.

The "get\_string\_token(...)" will handle multi-line strings - so it can increase the "line\_number", but only to reach the end of a multi-line string.

### Periods

The period character '.' is handled several different ways. This happens inside

Tokenizer :: get\_token(ref int line\_number, ref int char\_number)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Example | Tokenization Result | | | |
| .123 | 0.123 |  |  |  |
| "abc".count( | "abc" | . | count | ( |
| my\_str.count( | my\_str.count | ( |  |  |

Period at the start of a token, followed by a digit, is tokenized as part of a floating number.

Period at the start of a token, not followed by a digit, is made a token by itself, with TokenType = Period.

Period at in the middle of an identifier token is treated as part of that token. An alternative way to state this rule is that if a token starts with a letter, the code will scan to the right to look for the end of that token. Seeing a period does not count as the end of a token.

### Pattern matching

enum TokenType

   // These tokens are for pattern matching

AnyOne, ZeroOrMore, OneOrMore, TwoOrMore, NotNextToken, BracketExpr,

TokenList :: int find\_ending\_token(int start\_index)  
TokenList :: int find\_pattern(TokenType[] pattern, int start\_index, ...)

TokenList :: int find\_token(TokenType token\_type, int start\_index)

TokenList :: bool match\_pattern(TokenType[] pattern, int index)

**NotNextToken**

The combination of {NotNextToken, Def} will match any token except for {Def}.

## Parser

**File: Parser.cs**

**Operation**

The Tokenizer creates a set of "TokenList" objects. The parser turns these into "Statement" objects.

class Parser

List<TokenList>   
token\_lists

(from Tokenizer)

List<Statement>

statements

TokenList[]

intermediate\_code

### Intermediate Code

**Simplification rules**

|  |  |  |  |
| --- | --- | --- | --- |
| Pattern |  | Simplification | Handler Function |
| x += y x -= y x \*= y x /= y | 🡪 | x = x + y x = x - y x = x \* y x = x / y | simplify\_plus\_equal(...) |
| while x: z | 🡪 | while  while x: z | simplify\_while(...) |
| if: z elif: z else: z for: z while: z | 🡪 | if:  z ... while:  z | simplify\_single\_line\_if(...) |
| for i in x[1:3]: | 🡪 | $temp0 = x[1:3] for i in $temp0: | simplify\_for\_loop\_iterator(...) |
| x f() y, x ≠ def | 🡪 | f()  $temp0 = $return  x $temp0 y | simplify\_function(...) |
| x "str" . f ( y | 🡪 | $temp0 = "str"  x $temp0.f ( y | simplify\_string\_method\_call(...) |
| x s[::].f(y | 🡪 | $temp0 = s[::] x $temp0.f (y | simplify\_slice\_method\_call(...) |

**Rule matching implementation**

class Parser  
 static TokenType[][] simplification\_patterns;

// handlers for the simplification patterns

delegate void SimplificationHandler(LinkedListNode<TokenList> node, ...);

static SimplificationHandler[] simplification\_handlers;

The rules are encoding as TokenType[] patterns.

Each pattern has a handler function. For example, if the "simplification\_patterns[3]" is found to match, then "simplification\_handlers[3]" is called to simplify.

The pattern matching happens in check(...). The simplification functions are the simplify\_xxx(...) functions.

The patterns are divided into two groups, separated by the "repeating\_patterns" number. The "patterns[0]" through "patterns[repeating\_patterns-1]" are checked once.

Starting with "patterns[repeating\_patterns]", each line of code is examined from left to right. At each index location, ALL of the rules are checked.

Some configuration files use many independent scripts, meaning there will be multiple parser objects. That is why the "simplification\_patterns" and "simplification\_handlers" are static.

### Call Tree

Parser(List<TokenList> token\_lists)

|🡪 create\_intermediate\_code(...)

| |🡪 check(LinkedListNode<TokenList> node)  
 | |🡪 simplify\_plus\_equal(LinkedListNode<TokenList> node, ...)  
 | |  
 | |🡪 simplify\_while(LinkedListNode<TokenList> node, ...)  
 | | |🡪 simplify\_single\_line\_if(LinkedListNode<TokenList> node, ...)  
 | | |🡪 check(LinkedListNode<TokenList> node) ⭯  
 | |  
 | |🡪 simplify\_single\_line\_if(LinkedListNode<TokenList> node, ...)  
 | | |🡪 check(LinkedListNode<TokenList> node) ⭯  
 | |  
 | |🡪 simplify\_for\_loop\_iterator(LinkedListNode<TokenList> node, ...)  
 | | |🡪 check(LinkedListNode<TokenList> node) ⭯  
 | |

| |🡪 simplify\_function(LinkedListNode<TokenList> node, ...)

| | |🡪 check(LinkedListNode<TokenList> node) ⭯  
 | |  
 | |🡪 simplify\_string\_method\_call(LinkedListNode<TokenList> node, ...)  
 | |  
 | |🡪 simplify\_slice\_method\_call(LinkedListNode<TokenList> node, ...)

| |🡪 check(LinkedListNode<TokenList> node) ⭯  
 |  
 |🡪 construct\_statements(int start\_index, int end\_index, int? break\_addr, ...)   
 | new AssignStatement(...)  
 | new ConditionalStatement(...)  
 | ...  
 | new SliceAssignStatement(...)  
 |  
 |🡪 process\_def\_statement(ref int index)  
 | |🡪 int find\_next\_line\_with\_equal\_or\_less\_indentation(int index)  
 | |🡪 construct\_statements(int start\_index, int end\_index, ...) ⭯  
 |  
 |🡪 process\_for\_statement(ref int index)  
 | |🡪 int find\_next\_line\_with\_equal\_or\_less\_indentation(int index)  
 | |🡪 construct\_statements(int start\_index, int end\_index, ...) ⭯  
 |  
 |🡪 process\_if\_statement(ref int index, int? break\_addr, int? continue\_addr)  
 | |🡪 int find\_next\_line\_with\_equal\_or\_less\_indentation(int index)  
 | |🡪 construct\_statements(int start\_index, int end\_index, ...) ⭯  
 |

|🡪 process\_while\_statement(ref int index)  
 |🡪 int find\_next\_line\_with\_equal\_or\_less\_indentation(int index)  
 |🡪 construct\_statements(int start\_index, int end\_index, ...) ⭯

**Recursion**

The ⭯ means recursion.

The "check(...)" is being called recursively to handle the new statements that are generated by "simplify\_xxx(...)".

The "construct\_statements" is being called recursively to handle the body of "if", "while", and "for" statements.

### TempVar

**File: Parser.cs**

class TempVar

static string get\_var\_name(int indentation)

static void reset(int indentation)

The TempVar class generates the temporary variables used by the intermediate code. These temporary variables are: $temp0, $temp1, $temp2, and so on.

**Reusing temporary variables**

Each line of the original source code, meaning each "token\_list" BEFORE it is fed into the Parser, generates an independent block of intermediate code. This allows the temporary variables to be reused.

Inside Parser :: create\_intermediate\_code(...):

  foreach (var code in token\_lists)

  {

**TempVar.reset(...);**

    ...

    check(node);

  }

After each "reset(...)" call, each "TempVar.get\_var\_name(...)" will return $temp0, reusing the temporary variables.

**Hierarchical temporary variable reuse**

To support the "for" statement, the temporary variable "reset()" needs to happen with indentation taken into account. For reasoning, see the section "design 🡪 flow control 🡪 for".

The TempVar internally is a table that keeps track of what is the highest temporary variable number in use for each indentation level.

So one example might be:

|  |  |
| --- | --- |
| indentations[] | highest\_var\_numbers[] |
| 0 | 3 |
| 4 | 5 |

This means $temp0 ... $temp3 is being used by intermediate code that has an indentation level of 0, and $temp4 ... $temp5 is being used by intermediate code that has an indentation level of 4.

In this situation:

* get\_var\_name(indentation = 0) 🡪 $temp4
* get\_var\_name(indentation = 4) 🡪 $temp6
* reset(4) 🡪 second row of the table becomes: highest\_var\_numbers[0] = 3
* reset(0) 🡪 first row of the table: highest\_var\_numbers[0] = -1, second row of the table will be deleted.

**To test the TempVar using the interactive window**

TempVar.get\_var\_name(0) // $temp0  
TempVar.get\_var\_name(0) // $temp1  
TempVar.get\_var\_name(0) // $temp2

Write(TempVar.to\_string())

TempVar.get\_var\_name(4) // $temp3  
TempVar.get\_var\_name(4) // $temp4  
TempVar.get\_var\_name(4) // $temp5

TempVar.reset(4)  
Write(TempVar.to\_string())

TempVar.get\_var\_name(4) // $temp3  
TempVar.get\_var\_name(4)

Write(TempVar.to\_string())

### LabelAddresses

**File: Parser.cs**

class LabelAddresses

int allocate\_label()

void set\_label\_address(int label\_id, int address)

int get\_label\_address(int label\_id)

Internally "LabelAddresses" is "List<int> addresses".

The "label id" values are the indices.

The "label id" values are used as negative addresses. So if a "ConditionalStatement" contains:

TrueAddr = 15  
FalseAddr = -3

the "15" is a real address, while the "-3" really means label id "3".

When the Parser reach the end of the "if" statement, the parser will put in a real address at "addresses[3]".

When all statements have been generated, all the fake (negative) addresses will be resolved by looking up the "LabelAddresses" class.

## Statements

**File: Statements.cs**

**Statement**

interface Statement

|🡪 class AssignStatement : Statement  
 |🡪 class ConditionalStatement : Statement  
 |🡪 class FunctionArgAssignStatement : Statement

|🡪 class FunctionCallStatement : Statement  
 |🡪 class GlobalStatement : Statement  
 |🡪 class JumpStatement : Statement  
 |🡪 class ReturnStatement : Statement

The statement classes contain information. The execution code is elsewhere:

|  |  |
| --- | --- |
| class RunTime | Runs the Statement objects |
| class Expression | Runs the Expression objects that are inside the statements. |
| Value :: Value operate(...) | Runs the operators, like addition and subtraction. |
| Value :: Value call(...) | Runs the methods, like string.isnumeric(...). |

**Common to all statements**

interface Statement

{

  StatementType Type { get; }

  int Indentation { get; }

  int LineNumber { get; }

  void resolve\_addresses();

}

### Assign Statement

x = 1 + 2

class AssignStatement

string lvalue;

Expression expression;

class AssignStatement

lvalue = "x"

class Expression

ExpressionTerm

ExpressionTerm

ExpressionTerm

### Conditional Statement

This is generated by "if", "while", "for", ...

class ConditionalStatement

Expression expression;

int TrueAddr { get; private set; }

int FalseAddr { get; private set; }

### Function Arg Assign Statement

This is generated by function definitions.

def f(x, x2 = 123):

The "x" is being assigned the first argument, the "x2" is being assigned the second argument, with "123" as the default.

class FunctionArgAssignStatement

string lvalue

int position

Expression default\_expression

### Function Call Statement

f(x)

class FunctionCallStatement

string object\_name;  
 string function\_name;

FunctionArguments arguments

**Constructor call tree**

FunctionCallStatement(TokenList token\_list)

|🡪 parse\_function\_arguments(TokenList token\_list)

~~|🡪 parse\_slice(TokenList token\_list)~~

The constructor has two branches, one for f(...) style functions, and another (disabled branch) for str[2:4] style slice notation. At one point, the slice notation was implemented as a function call.

### Global Statement

global x, y

class GlobalStatement

string[] Symbols

### Jump Statement

This is generated by "if", "for", "while", ...

class JumpStatement

int Addr { get; private set; }

### Keyword Statement

del x  
del x[1:10]

class KeywordStatement : Statement

  public readonly Token keyword;

   // Use (at most) ONE of the following. The unused variables are null.

  public readonly string var\_name;

  public readonly SliceExpression slice\_expression;

   public readonly Expression expression;

### Return Statement

This is returning from a function.

class ReturnStatement

Expression expression

### Slice Assignment

x[1:3] = [1, 2, 3, 4, 5, 6]

class SliceAssignStatement : Statement

  public readonly SliceExpression lvalue;

   public readonly Expression expression;

## Expression

**File: Expression.cs**

**Expression**

class Expression

ExpressionTerm[] terms;

class Expression

ExpressionTerm

ExpressionTerm

ExpressionTerm

**ExpressionTerm**

class ExpressionTerm

Type type;

object value;

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| Original expression: | 3.4 | + | x |
| type | Value | Operator | Identifier |
| value | DynamicValue(3.4) | Operator class | string("x") |

**Expression Evaluation**

Expression evaluation is done using the "Shunting-yard" algorithm.

During construction time, the Tokens are processed into ExpressionTerm objects, and stored in reverse Polish notation (RPN) order.

The run time evaluates the expression by calling:

Expression::Value eval(SymbolTable symbol\_table)

**Construction Call Tree**

Expression(TokenList source, int start\_index, int end\_index)  
 |🡪 void process\_expression\_term(TokenList token\_list, ref int index,

| |🡪 bool is\_active\_operator(Token token)  
 |

|🡪 void pop\_op\_stack(...)

**Negation Handling**

The subtraction character has dual meaning.

It could mean subtraction, as in 4 - 1, in which case it's a binary operator. It could also mean negation, as in 4 + -1, in which case it's a unary operator.

During tokenization, the subtraction character maps to the symbol "TokenType.Subtract".

During Expression construction, this token is interpreted either as "OperatorType.Subtract" or "OperatorType.Negate".

The interpretation is based on the character preceding the subtraction symbol.

### Operator

**File: Expression.cs**

class Operator

OperatorType type;

int precedence()

**Construction**

All operators objects are constants - so there is only one "Add" operator object in the whole program.

"Operator" objects are constructed from "Token" objects.

There's a data structure for mapping from tokens to operators:

static Dictionary<TokenType, Operator> token\_to\_operator;

## Value

**File: value.cs**

**Value Interface**

interface Value : IEquatable<Value>

{

  ValueType Type { get; }

  Value operate(OperatorType op\_type, Value val2);

  Value call(string function\_name, FunctionArguments arguments,

    SymbolTable symbol\_table);

}

**Value Types**

interface Value

|🡪 class DynamicValue : Value  
 |🡪 class NoneValue : Value  
 |🡪 class RangeValue : Value

|🡪 class StringValue : Value

**operate(op\_type, val2)**

The exact meaning of the operate(...) function depends on the "op\_type".

This function is called by Expression::eval(...).

For unary operators, the "val2" parameter is null. The code inside Expression::eval(...) is:

if (op.type == OperatorType.Negate || op.type == OperatorType.Not

  || op.type == OperatorType.Bitwise\_Not)

{

  // unary operator cases  
 ...  
 // apply the operator  
 ...

var result = val1.operate(op.type, null);

For binary operators, the "val2" parameter is normally the parameter on the right-hand side. This means for the expression "1 + 2", the "2" is the "val2". The operator function call looks like:

DynamicValue(1).operate(ADD, DynamicValue(2))

The binary operator code inside Expression::eval(...) is:

// Usually it's val1.operate(op.type, val2)

// But invoke using val2 for some operators

if (op.type == OperatorType.In || op.type == OperatorType.Not\_In)

{

  // reverse operate() invoke

  result = val2.operate(op.type, val1);

}

else

  // standard operate() invoke

  result = val1.operate(op.type, val2);

So it's usually val1.operate(...) - meaning the "operate(...)" belongs to the object on the left hand side of the operator.

For the "in" and "not in" operation, the "operate(...)" function belong to the object on the right hand side.

**operate(...) return**

Return "null" for non-supported operations.

The "operate(...)" is called by "Expression::eval(...)", which will see the "null" return and throw an exception.

### Iteration Methods

To support iteration, the official methods are: **next\_element(), has\_next\_element(), and reset()**.

To speed things up, sometimes only the first character is checked. For example, in the "Range" object:

class RangeValue :: public Value call(string function\_name, ...)

{

  // full function names are: next\_element(), has\_next\_element(), and reset()

  // cheap acceleration - just check the first letter of "function\_name"

  if(function\_name[0] == 'n')

  {

    ...

  }

  else if (function\_name[0] == 'h')

  {

    ...

  }

  else if (function\_name[0] == 'r')

  {

    ...

  }

This "should" be okay because the user does not call these iteration methods directly. These method calls are generated when a "for" loop is processed into statements.

Also, note that this is NOT the way Python implement iteration.

The following "Python" script will work with this interpreter, but not with a real Python interpreter.

x = range(1, 5, 2)

print(x.has\_next\_element()) # True

print(x.next\_element()) # 1

print(x.next\_element()) # 3

print(x.next\_element()) # None

print(x.has\_next\_element()) # False

print("resetting x")

x.reset()

print(x.has\_next\_element()) # True

print(x.next\_element()) # 1

print(x.next\_element()) # 3

### DynamicValue

class DynamicValue

readonly dynamic value;

Dynamic Value operations are handled by

DynamicValue :: Value operate(OperatorType op\_type, Value val2)

var result = op\_handlers[(int)op\_type](value, y);  
  
 static DynamicValue()

op\_handlers[(int)OperatorType.Add] = (x, y) => x + y;

op\_handlers[(int)OperatorType.And] = (x, y) => x && y;

op\_handlers[(int)OperatorType.Bitwise\_And] = (x, y) => x & y;

op\_handlers[(int)OperatorType.Bitwise\_Not] = (x, y) => ~x;  
 ...

### NoneValue

This represents the "None" in Python.

class NoneValue  
 // This object is a constant. There is only one "NoneValue" object:

public readonly static NoneValue NONE;

### RangeValue

This represents the "range(...)" object in Python.

class RangeValue

int start, stop, step, current;

The RangeValue is made to support the "for" statement. It is not the same as the "range(...)" object in Python.

See the section "Value 🡪 Iteration Methods" for a script that can test the RangeValue directly, without the use of a "for" loop.

### StringValue

This represents a string in Python.

class StringValue

readonly string value;

**Methods**

static Value slice(string value, FunctionArguments arguments, ...)

static Value capitalize(string value, FunctionArguments arguments, ...)

static Value center(string value, FunctionArguments arguments, ...)

static Value count(string value, FunctionArguments arguments, ...)  
...

static Value zfill(string value, FunctionArguments arguments, ...)

**Method lookup table**

delegate Value Method(string value, FunctionArguments arguments, ...);

static Dictionary<string, Method> methods\_look\_up;

The StringValue class has large number of methods. To speed up the process of locating the method, a lookup table is used.

A script might have many StringValue classes, so the "methods\_look\_up" is static, such that there is only one copy of the lookup table. All of the methods are then forced to be static as well.

**Slice notation default differs from function default**

Slice notation defaults are slightly different than function call defaults. In both cases, missing arguments trigger the default. The difference is that in function calls, the missing argument is always at the end. The slice notation missing argument looks like:

str[1::2]

The missing argument is in the middle, between the 1 and the 2.

The second difference is that the default is not hard coded. The default value in the example above is the length of the string, which cannot be hardcoded ahead of time.

**Slice notation default handling implementation**

FunctionCallStatement constructor:

FunctionCallStatement(TokenList token\_list)

|🡪 void parse\_slice(TokenList token\_list)

{

// Handle the case where a default is being used, like x[:2]

arguments**.add\_null\_argument();**

}

StringValue slice(...):

StringValue :: Value slice(FunctionArguments arguments, SymbolTable symbol\_table)  
{

if (arguments.Count == 1)

  {

// Handle the single argument case.  
 }

// Code gets here if the [] notation has 2 or 3 arguments.

 int end = arguments.get\_int\_argument(1, null, **value.Length**, symbol\_table, ...);  
}

## List

**File: ListValue.cs**

**Class Relationships**

.eval()

ListExpression

ListValue

ListValue,  
StringValue

.eval()

SliceExpression

The general pattern is that an "Expression" ".eval()" produces a "Value":

Expression :: Value eval(SymbolTable symbol\_table)  
ListExpression :: ListValue eval(SymbolTable symbol\_table

SliceExpression :: Value eval(SymbolTable symbol\_table)

### ListExpression

Internally:

class ListExpression

Expression[] expressions;

Each "expression" then evaluate to a single item in a list.

**Capabilities Summary**

|  |  |
| --- | --- |
| Can construct from tokens | ListExpression(TokenList token\_list, ...) |
| Can be evaluated into "ListValue" | ListValue eval(SymbolTable symbol\_table) |

### ListValue

Internally:

class ListValue : Value  
 readonly IList<Value> list

// for iteration:

int current\_iter = 0;

The "ListValue" data is held in a "list". This is either a "List<Value>", or a "Slice" wrapper around "List<Value>".

**Capabilities Summary**

|  |  |
| --- | --- |
| Construct from List<Value>, as an empty list, or wrap a "Slice" object around an existing list. | ListValue(List<Value> list)  ListValue()  ListValue(ListValue list\_value, int user\_start, int user\_stop, int step) |
| Determine whether the internal "IList list" is a real list, or is a slice of a list. | bool is\_slice() |
| Shallow copying support - right now this is actually not needed. | static Value shallow\_copy(Value value)  ListValue get\_shallow\_copy()  Value get\_shallow\_copy(int index) |
| Looks like a C# list through implementation of commonly used properties. | Value this[int index]  int Count |
| Compare functions to support sorting. | static int compare\_function(Value x, Value y)  static int reverse\_compare\_function(Value x, Value y) |
| Method support | Value call(string function\_name, ...)  static ListValue()  {  methods\_look\_up.Add("append", append);  ...  methods\_look\_up.Add("sort", sort); } |

**Shallow Copy**

List copy in Python defaults to shallow copy. However, this is automatically happening, and all the "shallow\_copy" related methods currently does nothing.

The key reason is that the DynamicValue class is immutable. In an efficient implementation,

x = x + 1

will store "x+1" into the object "x" - no new object is created.

But in the current inefficient, but easy to understand implementation, "x+1" is creating a new object, which is then stored as "x" in the symbol table.

In the case of a list:

|  |  |
| --- | --- |
| list1 = [1, 2, 3] |  |
| list2 = list1[:] | list2[0] and list1[0] both point to "1" |
| list2[0] = list2[0] + 1 | list2[0] now points to a NEW object, "2" |

As a result, operating on list2[0] does not effect list1[0].

If the Expression::eval() were to be made more efficient, such that "list2[0] + 1" does not generate a new object, then shallow copy needs to duplicate the "1" during copy.

**compare\_function(...), reverse\_compare\_function(...)**

Comparison functions are necessary for C#'s List<Value>::sort(...).

These function will trigger the "Value::operate(...)", and will use "OperatorType.Equal" and "OperatorType.Greater" to implement the comparison.

**is\_slice(), cast\_to\_value\_list()**

The list data is contained inside:

IList<Value> list

This "list" can either be "List<Value>" or "Slice". The "List<Value>" contains the real data while the "Slice" remaps index to access the underlying "List<Value>".

Some methods only apply if the "list" is real data. These methods will use "is\_slice()" and "cast\_to\_value\_list()" to check whether the underlying "list" is real data.

**ToString()**

There is some special formatting, so to match Python output. For example, "\n" needs to appear on screen as "\n", not as a newline.

### Slice Expression

Internally:

class SliceExpression

{

  // the source of the slice is ONE of the following (the unused variables are null)

  string var\_name;

  StringValue str\_literal;

  SliceExpression inner\_slice\_expr;

  // start, stop, step

  readonly int num\_arguments = 0; // to differentiate s[2::] versus s[2]

**public readonly Expression start\_expression;**

**public readonly Expression stop\_expression;**

**public readonly Expression step\_expression;**

The "SliceExpression" contain Expression objects for the slice notation [start : stop : step].

**Capabilities**

|  |  |
| --- | --- |
| Construct from tokens | SliceExpression(TokenList token\_list, int start\_index, int end\_index) |
| Evaluates to "ListValue" or "StringValue" | Value eval(SymbolTable symbol\_table) |
| Assign a value to this slice | void assign(Value value, SymbolTable symbol\_table) |
| Delete this slice | void del(SymbolTable symbol\_table) |

**eval(...) call tree**

SliceExpression :: Value eval\_list\_slice(ListValue value, ...)  
 |🡪 StringValue eval\_string\_slice(StringValue str\_value, ...)  
 | |🡪 int eval\_int\_expression(Expression expression, ...)  
 | |🡪 (int, int, int) get\_slice\_parameters(int length, ...)  
 |

|🡪 Value eval\_list\_slice(ListValue value, SymbolTable symbol\_table)

|🡪 int eval\_int\_expression(Expression expression, ...)

|🡪 (int, int, int) get\_slice\_parameters(int length, ...

**assign(...) call tree**

SliceExpression :: void assign(Value value, SymbolTable symbol\_table)

|🡪 int eval\_int\_expression(Expression expression, ...)  
 |🡪 (int, int, int) get\_slice\_parameters(int length, ...)  
 |🡪 void insert\_value(IList<Value> list, Value new\_value, int start, int stop)  
 |🡪 void assign\_items(IList<Value> list, Value new\_items, int start, ...)

### Slice

Internally:

class Slice : IList<Value>

  IList<Value> list;

   int start, stop, step;

A "Slice" class remap the index being used to access the "list".

public Value this[int index]

{

  get { return list[real\_index(index)]; }

  set

  {

    list[real\_index(index)] = value;

  }

}

So the user of a "Slice" class think they are accessing the list using indices (1, 2, 3, ...), but they are really using the underlying data with indices (real\_index(1), real\_index(2), real\_index(3), ...).

The computation of "real\_index":

int real\_index(int index)

{

  int real\_index = start + index \* step;  
 ... error checking ...  
  return real\_index;

}

**SliceEnumerator**

This class allows the "Slice" class to be used with C# foreach loops.

## Dictionary

**File: DictionaryValue.cs**

**Class Relationships**

.keys()  
.values()

.eval()

EnumeratorValue

DictionaryValue

DictionaryExpression

**DictionaryExpression**

class DictionaryExpression

  Expression[] key\_expressions;

  Expression[] value\_expressions;

The "DictionaryExpression" class is constructed from source code.

DictionaryExpression(TokenList token\_list, int start\_index, int end\_index)

At run time, the ".eval()" method will produce a "DictionaryValue".

DictionaryValue eval(SymbolTable symbol\_table)

**DictionaryValue**

class DictionaryValue : Value

  public readonly Dictionary<Value, Value> dict;

Like other "Value" classes this one has "Equals(...)", "call(...)", and "operate(...)".

The dictionary's ".keys()" and ".values()" methods produce "EnumeratorValue" objects.

static EnumeratorValue keys(Dictionary<Value, Value> dict, ...)

static EnumeratorValue values(Dictionary<Value, Value> dict, ...)

**EnumeratorValue**

class EnumeratorValue : Value

  public readonly IEnumerator<Value> enumerator;

The "EnumeratorValue" class implements "next\_element()", "has\_next\_element()", and "reset()" methods, so the Python code can iterate over the .Net based "IEnumerator".

public Value call(string function\_name, FunctionArguments arguments, ...)

{

if(function\_name[0] == 'n')

{  
 ...

}

else if (function\_name[0] == 'h')

{  
 ...

}

else if (function\_name[0] == 'r')

{  
 ...

}

**SliceExpression**

Dictionary read and write expressions are implemented via "SliceExpression" objects.

In:

x = d['one']

the "x = d['one']" is turned into a "SliceExpression". At run time, a read from the "d" dictionary is done via:

SliceExpression :: Value eval(SymbolTable symbol\_table)

|🡪 Value eval\_dict\_value(DictionaryValue value, SymbolTable symbol\_table)

In:

d['one'] = 1

The "d['one']" is once again a slice expression. At run time, the assignment of "1" into dictionary "d" is done via:

SliceExpression :: void assign(Value value, SymbolTable symbol\_table)

## RunTime

**File: RunTime.cs**

class RunTime

List<Statement> statements;

int prog\_counter = 0;

**Call tree for running statements**

RunTime :: bool run(int? instructions\_to\_run = null)  
 statement\_handlers[(int)statement.Type](statement);

|🡪 void run\_assign\_statement(Statement statement)  
 |🡪 void run\_conditional\_statement(Statement statement)

| ...  
 |🡪 void run\_return\_statement(Statement statement)

The "statement\_handlers" is a jump table to various handler functions.

**Function Call Stack**

class FunctionCall

  int return\_addr;

  FunctionArguments arguments;

class RunTime

Stack<FunctionCall> function\_call\_stack

used in:

void run\_function\_call\_statement(Statement statement)  
void run\_return\_statement(Statement statement)

The "run\_function\_call\_statement(...)" handles several kind of function calls:

* built-in functions - these are written in C#
* built-in methods - like for the built-in string class. These are also in C#
* user defined functions - these are in Python, via "def f():"

When making a call to a user defined function, a "FunctionCall" object is created and pushed onto the "function\_call\_stack".

Separately, the "symbol\_table" is told to create a new symbol table for use with the function:

symbol\_table.enter\_function\_call();

When exiting the user defined function, the "run\_return\_statement(...)" unwinds the stack.

### Symbol Table

The symbol table maps a (string) variable name to a "Value" object.

class SymbolTable

Value **get(**string symbol, ...)

void **store(**string symbol, Value value)

The symbol table also supports variable scoping. When entering and exiting functions, call:

class SymbolTable

void **enter\_function\_call()**

void **exit\_function\_call()**

Internally:

class SymbolTable

Dictionary<string, Value> **global\_symbol\_table**

Stack<Dictionary<string, Value>> **local\_symbol\_table\_stack**

There's a global symbol table, and a stack of local symbol tables to provide local scoping for function calls.

Python code inside functions refer to the global variable via the "global" statement. This statement creates an entry in:

class SymbolTable  
 Stack<HashSet<string>> **global\_symbols\_stack**

During a function call, only the top "HashSet" in the "global\_symbols\_stack" is in effect.

If a variable name is mentioned in the "HashSet", then that variable name refers to a global variable.

**object recycling optimizations**

The "get\_eval\_stack()" and "local\_symbol\_table\_buffer" are attempts to speed things up by reducing the number of object allocations.

**get\_eval\_stack(), return\_eval\_stack()**

These are used by the "Expression::eval(...)".

The goal is to slightly speed up Expression::eval(...).

The "Expression::eval(...)" algorithm uses a stack to evaluate RPN (Reverse Polish Notation) expressions. The most basic approach is to allocate a new stack for each "eval(...)" call. To speed up things slightly, the stack being used is recycled between eval(...) calls.

**local\_symbol\_table\_buffer, global\_symbols\_buffer**

When a function runs, it creates a new local symbol table, for the local variables in the function. There's also a new hash set to store variables that are named as "global".

The goal of these buffers is to avoid repeated variable allocation when making a function call, slightly speeding up the "SymbolTable::enter\_function\_call()" and "SymbolTable::exit\_function\_call()" functions.

### Function Arguments

**File: RunTime.cs**

The "FunctionArguments" class is a container for the arguments when making function calls. It has methods to add and retrieve arguments.

class FunctionArguments

List<Expression> positional\_args;

Dictionary<string, Expression> named\_args;

void add\_argument(TokenList token\_list, int start\_index, int end\_index)

void add\_value\_argument(Value value, int line\_number)

void check\_num\_args(int n)

void check\_num\_args\_minimum(int n)

void check\_num\_args\_maximum(int n)

int get\_int\_argument(int? arg\_position, string arg\_name, ...)

string get\_string\_argument(int? arg\_position, string arg\_name, ...)  
 Value get\_value\_argument(int? arg\_position, string arg\_name, ...)

List<Value> get\_list\_arguments(int arg\_position, SymbolTable symbol\_table)

### Built In Functions

**File: BuiltInFunctions.cs**

The "BuiltInFunctions" (plural) class contains multiple "BuiltInFunction" (singular) definitions.

class BuiltInFunctions

delegate Value BuiltInFunction(FunctionArguments arguments);

Dictionary<string, BuiltInFunction> function\_lookup

Value len(FunctionArguments arguments)  
 ...  
 Value str(FunctionArguments arguments)

**Registration and Execution**

During construction of "BuiltInFunctions", the "print()" is added to the "function\_lookup" table.

function\_lookup.Add("print", print);

During execution, the "print()" is triggered via

BuiltInFunctions :: Value run(FunctionCallStatement function\_call)

## Differences from Python

**Differences due to the .Net Dynamic Runtime**

The class "DynamicValue" is relying on the .Net dynamic runtime libraries to handle operators. The .Net runtime differs from the Python run time in some operations.

Example:

|  |  |
| --- | --- |
| Code | print(1 + **True**) |
| Python | 2 |
| This interpreter | Exception:  The evaluation of 1 Add True failed. Details: Operator '+' cannot be applied to operands of type 'int' and 'bool'' |

**String auto conversion**

Python does not auto convert strings. The "str(...)" is necessary to trigger string conversion in many common cases.

C# as a language tends to auto convert strings. This interpreter will sometime auto convert strings.

Example:

|  |  |
| --- | --- |
| Code | s = "\*".join(["a", 2, 'c']) print(s) |
| Python | TypeError: sequence item 1: expected str instance, int found |
| This interpreter | a\*2\*c |

**No copy on write implementation**

For loop over a slice object, or calling method on a slice object, should not duplicate the original list, so to save memory.

Yet, it's clear that Python is doing some kind of copying, sometimes. For example:

x = [1, 2, 3, 4]  
**for** i **in** x[1:]:  
 x[1] = 10  
 x[2] = 10  
 x[3] = 10  
 print(i, end = ' ')  
  
print() # 2 3 4  
print(x) # [1, 10, 10, 10]

The [2, 3, 4] are original list values, even though the content of the list has been immediately changed in the for loop.

The guess is that Python has a sophisticated "copy on write" feature, where it makes copy of the underlying list as needed.

This copy on write is not implemented in this interpreter.

|  |  |
| --- | --- |
| Code | x = [1, 2, 3, 4] **for** i **in** x[1:]:  x[1] = 10  x[2] = 10  x[3] = 10  print(i, end = ' ')  print() # 2 3 4 print(x) # [1, 10, 10, 10] |
| Python | 2 3 4  [1, 10, 10, 10] |
| This interpreter | 2 10 10  [1, 10, 10, 10] |

To get the "2 3 4" behavior, manually make a copy:

x = [1, 2, 3, 4]  
x\_copy = x[1:]  
**for** i **in** x\_copy:  
 x[1] = 10  
 x[2] = 10  
 x[3] = 10  
 print(i, end = ' ')  
  
print() # 2 3 4  
print(x) # [1, 10, 10, 10]

**Python silent failures (ignore) on encountering certain slice or "slice of a slice" code**

Python (3) ignores certain "slice" or "slice of a slice" code - nothing seems to happen, but there is no complaint either.

This interpreter prefers to either do something or complain.

A do something example (item assignment to slice of a slice):

|  |  |
| --- | --- |
| Code | x = [1, 2, 3, 4] x[1:4][0] = 100 print(x) |
| Python | [1, 2, 3, 4] |
| This interpreter | [1, 100, 3, 4] |

A complaining example (invalid method call on a slice):

|  |  |
| --- | --- |
| Code | x = [1, 2, 3, 4] x[1:].append(100)  print(x) |
| Python | [1, 2, 3, 4] |
| This interpreter | Error details: The method append() is only supported for a real list, not the slice of a real list.' |

A complaining example (list assignment to a slice of a slice):

|  |  |
| --- | --- |
| Code | x = [1, 2, 3, 4] x[0:3][1:3] = [100, 101] print(x) |
| Python | [1, 2, 3, 4] |
| This interpreter | Error details: Inserting a value into the slice of a slice is not supported.' |

A complaining example (deleting a slice of a slice):

|  |  |
| --- | --- |
| Code | x = [1, 2, 3, 4, 5, 6] **del** x[1:][1:3]  print(x) |
| Python | [1, 2, 3, 4, 5, 6] |
| This interpreter | Error details: del a slice of a slice is not supported. |

**min(), max() for multiple lists**

In this case, the Python implementation is to compare just the first element of the list, which doesn't seem that useful.

|  |  |
| --- | --- |
| Code | print(max([1, 100, 101], [2, -100, -101])) print(min([1, 100, 101], [2, -100, -101])) |
| Python | [2, -100, -101]  [1, 100, 101] |
| This interpreter | 101  -101 |

**range() accepts doubles**

It's convenient and more readable to write "1e6" instead of "1000000".

The range function of this interpreter rounds "double" to "int".

|  |  |
| --- | --- |
| Code | x = 0 **for** i **in** range(0, 1e6, 1):  x += 5  print(x) |
| Python | TypeError: 'float' object cannot be interpreted as an integer |
| This interpreter | 5000000 |

# Extension Examples

**Files**

* **Program.cs**
* **extension\_examples.py**

**Variable Initialization**

C# injects values into the symbol table.

class ExtensionExample

run\_time.symbol\_table.store("a", new DynamicValue(1));

run\_time.symbol\_table.store("b", new DynamicValue(2));

Then the Python side can use these variables without initialization.

print(a) # 1  
print(b) # 2  
print(a + b) # 3

**User defined function**

C# side defines a function that adds two numbers:

Value user\_add(FunctionArguments arguments, SymbolTable symbol\_table)

{

  arguments.check\_num\_args(2);

  var val1 = arguments.get\_double\_argument(0, null, null, symbol\_table);

  var val2 = arguments.get\_double\_argument(1, null, null, symbol\_table);

  return new DynamicValue(val1 + val2);

}

C# side then registers this function under the name "user\_add" with the run time:

run\_time.add\_function("user\_add", user\_add);

Python side calls the function:

x = user\_add(1.3, 2.4)  
print(x) # 2.7

**Object based programming**

C# side defines the class:

class UserSum : Value

{

  double sum = 0;

  public UserSum(double start)

  {

    sum = start;

  }

  public Value call(string function\_name, FunctionArguments arguments, SymbolTable symbol\_table)

  {

    if (function\_name.Equals("add"))

    {

      arguments.check\_num\_args(1);

      var val1 = arguments.get\_double\_argument(0, null, null, symbol\_table);

      sum += val1;

      return NoneValue.NONE;

    }

    else if (function\_name.Equals("get"))

    {

      arguments.check\_num\_args(0);

      return new DynamicValue(sum);

    }

    return null;

  }

C# side also needs to define a Python constructor function for the class:

Value UserSum(FunctionArguments arguments, SymbolTable symbol\_table)

{

  arguments.check\_num\_args(1);

  var start = arguments.get\_double\_argument(0, null, null, symbol\_table);

  return new UserSum(start);

}

As before, C# side needs to register this function with the run time:

run\_time.add\_function("UserSum", UserSum);

Python side can use the class:

user\_sum = UserSum(15)  
user\_sum.add(5)  
  
x = user\_sum.get()  
print(x) # 20