Documentation: 'Simple' Interpreter in Python

Philipp Kochanski, Joshua Schmidt

September 19, 2016

Abstract

The purpose of this project was to extend the 'Simple' programming language. During the semester we have build our own implementation of the object model and interpreter, which were both based on the previously provided 'Simple' parser, lexer and the AST model. The below described extension provides support for additional data types (boolean, float, string, list, map/dict), as well as related built-in functions/operations that are commonly available when working with these types. As a starting point for this project we have decided to use our own implementation of the interpreter, instead of the one that was provided to us at the end of the course exercises.

1 Changes

Starting with the project, as of exercise 10, there were several interpreter components that we had to modify in order to implement the requested data types. The below description for each component should serve as a short overview on what changes were necessary. To not blow out of proportion this documentation, we will focus on the most important changes and we will not discuss every new line of code. A full list of changes can be obtained following our git commit history.

1.1 Lexer

Lexer changes were limited to the addition of new regular expressions and tokens. In order to support the Float data type, we had to modify the existing Integer regular expression so that it does not intersect with the new Float regular expression. Additional regular expressions were needed for List and Dictionary where we had to add new tokens for square and curly brackets (ListOpenBracket, ListCloseBracket, MapOpenBracket ,MapCloseBracket). Furthermore, we added a regular expression and token for Booleans to handle **True** and **False** statements.

1.2 Parser

Following the lexer changes, we had to modify the 'Simple' parser to include new parser production rules. Here for each additional data type a new basic expression production rule was created that was responsible for returning the proper AST node construct.

1.3 Abstract Syntax Tree

In order to connect the tokens provided by the Lexer within an AST, we also needed to create new AST node definitions. Therefore, we have added 6 new literals (FloatLiteral, StringLiteral, BoolLiteral, KeyValueLiteral, ListLiteral, DictLiteral).

1.4 Object model

Implementing and working with the new 'Simple' data types also required changes to the object model. Same as before, we added 6 new objects, one for each new data type. All objects extend the 'W_NormalObject' prototype and include additional methods that allow certain object interaction needed for the interpreter code.

1.5 Interpreter

Lastly there were major changes to the interpreter code itself. Besides defining additional 'eval' cases, it was also necessary to modify the 'eval_MethodCall' and 'eval_PrimitiveMethodCall' definitions. Especially inside 'eval_PrimitiveMethodCall' we added code to handle built-ins for each new data type.

1.6 Built-ins

More information regarding added default built-ins can be found in [3]

2 New Data Types

2.1 Integer/Float

The new 'W_Float' type follows the previously available 'W_Integer' definition. It uses all 'W_Integer' supported arithmetic operations (built-ins) and allows for operations with mixed typed operands eg. it is possible to add an integer and a float without explicit type conversion of operands.

Integer example: *i = 1337* or *i = -666*
Float example: *f = 1.337* or *f = -6.66*

Note that the result of arithmetic operations with mixed typed operands will

always be of type 'W_Float'. In a scenario where explicit type conversion is needed, we have also included functions that parse integer to float, integer to string, floats to string and vice versa. Additional functions, for example comparison operations, can be found in the 'Built-ins' section [3.1].

2.2 Boolean

In order to write more complex programs with combined logic constraints for example within if statements we implemented the 'W_Boolean' object.

Boolean example: $t = True$ or $f = False$

This type provides several built-in methods yielding the common logical operators. The semantics are like `b1 and(b2)` where b2 can be another nested method returning a Boolean value. Examples can be found in the "Built-ins" section below.

2.3 String

The string object 'W_String' is validated True if the strings' value is not empty. Following Python style, strings can be written using single and/or double quotation marks.

String example: *s1 = "string1"* or *s2 = 'string1'*

This type provides some additional built-in methods to be able to combine, modify or compare several strings.

2.4 List

The new list object 'W_List' follows Python logic. It supports any given element (list of lists/dicts/strings/ints/floats/booleans) as well as lists with mixed elements.

List example: *list = [1,2.5,"string",[1,2],{1:2},True]*

We have also provided several built-in functions [3.4] that allow inspection and modification of lists.

2.5 Dictionary

The dictionary object 'W_Dict' is internally implemented using a Python dictionary. Relating to the semantics of a dictionary we decided to represent each element as a key-value tuple, the same way it is done in Python, i.e. using the form `key:value` for its elements.

Dictionary example: *dict = {1:2, 1.0:2.0, "key":"value", 2:[1], 3:{1:2}}*

As can be seen, lists and dictionaries may not be used as keys. Therefore, only 'W_String', 'W_Integer' and 'W_Float' objects are considered valid keys for a dictionary entry (as it is in Python).

Furthermore, we have decided to allow mixed types of keys in a single dictionary. Implemented built-ins can be found in the 'Built-ins' section [3.5] below.

2.6 Key-Value Tuple

Since we have decided to implement dictionaries like it is done in Python, i.e. using the semantic `key:value` for the elements, we need to provide such an object. Essentially a W_KeyValue object is just a wrapper for two values semantically split by a colon. Like described in [2.5] valid keys are of the type 'W_String', 'W_Integer' or 'W_Float'. This object does not provide any built-in methods because it is just used within dictionaries.

3 Built-ins

In addition to the above mentioned data types, we have also added several built-in functions to enrich available functionality. Built-ins are either directly written using Python (see interpreter code - eval_PrimitiveMethodCall) or by using already available "Simple" built-in functions and constructs. We have also provided examples as well as a test file (test_doc_examples.py) that demonstrates how built-ins are used.

3.1 Integer/Float

The result of arithmetic operations where mixed numeric types are used eg. int and float, will always be of type float.

- add(x): Add a number x (int/float)
  Example: *sum = 5 add(5)*
- sub(x): Substract a number x (int/float)
  Example: *dif = 5 sub(5)*
- mul(x): Multiply with number x (int/float)
  Example: *prod = 5 mul(5)*
- div(x): Divide with number x (int/float)
  Example: *quot = 5 div(5)*
- mod(x): Calculate the number modulo x
  Example: *mod = 12 mod(2)*

- sqrt(x): Calculate the number's square root - returns a float
  Example: *root = 12 sqrt*
- equals(x): '=' operator (for int/float) - returns True/False
  Example: *s = 5 equals(5)*
- less_than(x): '<' operator (for int/float) - returns True/False
  Example: *s = 5 less_than(6)*
- less_equal(x): '<=' operator (for int/float) - returns True/False
  Example: *s = 5 less_equal(5)*
- greater_than(x): '>' operator (for int/float) - returns True/False
  Example: *s = 6 greater_than(5)*
- greater_equal(x): '>=' operator (for int/float) - returns True/False
  Example: *s = 5 greater_equal(5)*

3.2 String

- len: Returns length of string
  Example: *length = '01234' len*
- append(x): Append string x to string (appending a number does not convert number to string implicitly)
  Example: *appended = '01234' append('56789')*
- reverse: Returns reversed version of string - keeps original string untouched
  Example: *reversed = '0123456789' reverse*
- equals(x): Checks whether two strings are equal - returns boolean (True/False)
  Example: *eq = '0123456789' equals('9876543210')*
- substring(x,y): Returns substring starting at x and ending before y
  Example: *sub = "substring" substring(0,3)*

3.3 Boolean

- not: If the left side is False, then True, else False
  Example: *b = True not*
- and(x): If the left side is False, then False, else x
  Example: *b = True and(False)*

- or(x): If the left side is False, then x, else True

  Example: *b = True or(False)*

- nand(x): negation of 'and' operation

  Example: *b = True nand(False)*

- nor(x): negation of 'or' operation

  Example: *b = True nor(False)*

- xor(x): exclusive or - True if one side only is true - otherwise False

  Example: *b = True xor(True)*

- xnor(x): exclusive 'nor' operation

  Example: *b = True xnor(False)*

- impl(x): Implication '⇒'

  Example: *b = True impl(True)*

- equals(x): Simple comparison whether left sind and x are equal

  Example: *b = True equals(True)*

3.4 List

- add(x): Add element x to (end of) list

  Example:
  l = [1,2,3]
  l add(4)

- insert(x,y): Insert element y to list at position x - moves existing elements of list one position to the right. If position does not exist, new element is simply added to the end of the list.

  Example:
  l = [1,3]
  l insert(1,2)

- replace(x,y): Replace element at position x with element y. If posiition does not exist, new element is simply added to the end of the list.

  Example:
  l = [1,3]
  l replace(1,2)

- del(x): Remove element at pos x from list

  Example:
  l = [1]
  l del(0)

- get(x): Get element at pos x from list

  Example: *e = [1,2,3] get(0)*

- len: Get length of list

  Example: *length = [1,2,3] len*

- reverse: Returns reversed version of list - keeps original list untouched

  Example: *r = [1,2,3] reverse*

- oreverse: Returns reversed version of list - original list is modified

  Example: *r = [1,2,3] oreverse*

- extend(x): Extends list by adding elements from x to left side (basically merge of two lists)

  Example: *l = [1,2,3] extend([4,5,6])*

- clear: Clears list/Removes all stored elements

  Example:
  l = [1,2,3]
  l clear

3.5 Dictionary

- add(key,value): Adds a key-value tuple to the dictionary - replaces existing entry if keys are identical

  Example:
  map = {'a':1}
  map add('b',2)

- del(key): Delete a tuple from the dictionary with given key.

  Example:
  map = {'a':1}
  map del('a')

- get(key): Get a value from the dictionary for a given key.

  Example: *value = {'a':1,'b':2} get('b')*

- get_keys: Returns a 'W_List' of all keys from the dictionary.

  Example: *keys = {'a':1,'b':2} get_keys*

- contains: Checks whether a dictionary contains a given key, returns True or False.

  Example: *check = {'a':1,'b':2} contains('a')*

- len: Returns the length of the dictionary, i.e. the amount of stored key-value tuples.

  Example: *length = {'a':1,'b':2} len*

3.6 Mixed

- to_int(x): Convert given float or string (if possible) to integer

  Example: *i = to_int(2.5)*

- to_float(x): Convert given integer or string (if possible) to float

  Example: *f = to_float("2.5")*

- to_str(x): Convert given integer or float to string

  Example: *s = to_str(1.337)*

- ceil(x): Returns the ceiling of x, the smallest integer greater than or equal to x.

  Example: *ce = ceil(2.6)*

- floor(x): Returns the floor of x, the largest integer less than or equal to x.

  Example: *fl = floor(2.6)*

- s_range(x): Simple range - returns a list where first element is 0 and last element is x

  Example: *l = s_range(5)*

- e_range(x,y): Extended range - returns a list where first element is x and last element is y

  Example: *l = e_range(5,10)*

- fibonacci(x): Returns the x-th number from the Fibonacci sequence

  Example: *fib = fibonacci(8)*

- gcd(x,y): Returns the greatest common divisor of x and y using the euclidian algorithm

  Example: *divisor = gcd(12,144)*

- isPrime(x): Checks if the given number is a prime number

  Example: *prime = isPrime(13)*

4 Conclusion

The mentioned extensions to the interpreter were implemented to provide the, in our opinion, most important language features that should be available before implementing additional extensions like a graphical user interface, coroutines or network support. We consider the current state of the interpreter as quite mature. The provided features make it possible to write reasonable programs using the 'Simple' programming language.