Implementing Strings in a Python Compiler

Compiler Construction Spring 2023 Final Project

Matt King, Will Snider, Collin Graham, Tucker Travins

ACM Reference Format:
Matt King, Will Snider, Collin Graham, Tucker Travins. 2023. Implementing Strings in a Python Compiler: Compiler Construction Spring 2023 Final Project. In *Proceedings of ACM Conference (Conference'23).* ACM, Boulder, CO, USA, 4 pages. https://doi.org/10.1145/1234567.1234567

Abstract

The Python to x86 compiler outlined in [6] is a solid foundation for a Python-based compiler. To continue expanding this implementation, we added support for strings, string slicing, and list slice assignment. The original scope of our project included a string implementation that involved building strings off of a char datatype. However, we soon discovered that only using a string datatype (with chars as just a single-character string) was a better path, so we switched to a big-type approach instead. In order to make up for the shrinking scope, and to continue to explore interesting language properties, we further augmented the compiler to also include list slice assignment (altering multiple indices in a list at once).

1 Background, Related Work, and Motivation

Strings have been an important part of programming languages for over 50 years, as they represent an important way for humans (who speak and write in natural language) to interact with computers in a more natural way than machine code, which only includes numbers. According to Sammet [5], strings were first introduced in the COMIT language [1], which was developed at Chicago and MIT on IBM computers.

This led to SNOBOL [2], a string-oriented language developed at Bell Labs in the 1960s. The unique feature of this language was that it was the first one to have strings as first-class data types, similar to how functions are treated in modern Python. SNOBOL also introduced string methods such as concatenation, and was the basis for future string uses such as regular expressions. The utility of SNOBOL was eventually replaced by languages such as Perl, that integrated string functionality as just a single part of a multi-function language. Ronald Morrison wrote a paper in 1982 [4] discussing the highlights of the datatype at the time, and how they were used in languages of that period, such as Algol, SNOBOL, and Fortran. He defines strings as similar to a vector, only every element must have the same type, and the methods are different to that of a vector. This is different to the common definition used today, in which a string generally must contain only characters, however his paper does focus on strings of characters. Thus, the defining feature of a string is not necessarily what it contains (characters) but what operations are available. Morrison also references a previous article by R. J. W. Housden [3], who shared the same sentiments: that strings aren't the same as vectors due specifically to their operations, mostly relating to substrings and extraction. Housden and Morrison describe the 8 main operations of a string as:

Creation: enclosing a literal in quotes and assigning it to a variable name.
`x = "abcdef"`

Concatenation: placing a string on the end of another string.
`x = "abc" + "def"`
x becomes "abcdef".

Segment Extraction: taking part of an existing string to make a new one.
`x = "abcdef"[0:3]`
x becomes "abc".

Substring Search: Locating a substring within a string and returning the index of that substring.
`x = "abcdef".find("cd")`
x becomes 2.

Comparison: Comparing strings for which comes alphabetically first (this string would be "less than" the other).
`x = "abc" < "def"`
x becomes True.

Substring Deletion/Replacement: Changing substrings within a string, but keeping the same object reference.
```
x = "abcdef"
x[0] = "z"
```
x becomes "zbcdef".
Note: This functionality is not often available in progamming language implementation of strings, as it is sometimes preferable for strings to be simple objects (instead of vector-like objects) and are thus immutable.

Conference'23, May 2023, Boulder, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1234567.1234567

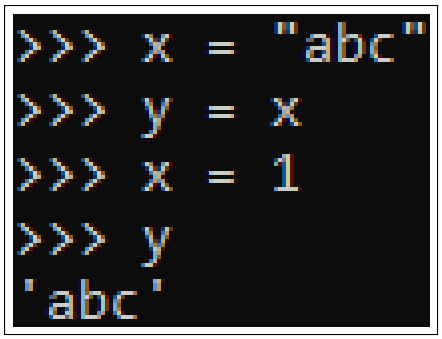


Figure 1. Python Aliasing

String Transput: Input and output of strings with terminals and files.
`print("abcdef")`

Length Interrogation: Dynamically obtaining the length of a string.
`x = len("abcdef")`
x becomes 6.

Obviously these are not commandments that must be followed by every succeeding language, so many languages have slightly diverged from these definitions. Python does not have substring deletion/replacement (strings are immutable), but new strings can be made using substrings of other strings. This is unique from lists (Python's implementation of a vector-like object), which allow sublists to be deleted and replaced while only working on a single object. Treating strings as a reference or a value is something that each programming language has to choose between (or be like Python and not let them be edited at all, and only allow copies to be made, not aliases (seen in Figure 1).

While editing strings through the removal or replacement of substrings is not possible in Python, the extraction of substrings is a very important operation of strings, and its main use as an alternative to vectors. This is the reason slicing is going to be implemented in this subset of Python.

2 Notations

The grammar for the implementation of strings in Python can be seen in Figure 1. The language is a superset of P2. This grammar allows for the implementation of strings, slicing, and list slice assignment. This is the standard Backus-Naur form used in class.

3 String and Slicing Implementation

The first thing we added to the compiler was support for strings. Our approach was to treat strings as a new type of `big_pyobj`, so we created a new `big_type_tag` for strings. We added a `struct` for strings that that includes a data pointer and length, and added this struct to the union field in the `big_pyobj struct`. This approach allows to treat strings

```
expression   ::=   "\"" string "\""
               |   slice

string       ::=   word
               |   char
               |   nullchar

word         ::=   char char
               |   char word
               |   word char

slice        ::=   expression "[" expression ":" expression ":" expression "]"

               |   expression "[" expression ":" expression "]"
               |   expression "[" expression ":" expression ":" "]"
               |   expression "[" ":" expression ":" expression "]"
               |   expression "[" expression ":" ":" expression "]"

               |   expression "[" expression ":" "]"
               |   expression "[" expression ":" ":" "]"
               |   expression "[" ":" expression "]"
               |   expression "[" ":" ":" expression "]"
               |   expression "[" ":" expression ":" "]"

               |   expression "[" ":" ":" "]"
               |   expression "[" ":" "]"
```

Figure 2. Grammar Extension

as a list of numbers that are stored on the heap. We added run-time functions for strings based on the existing functions for lists. This included modifying the `print_any` function to call a `print_string` function, adding an `is_string` function, and adding a `create_string` function to help with boxing. We then added support for string subscription and slicing.

3.1 Lex & Parse

We used Python's abstract syntax tree module, `ast`, to lex and parse the input program. The function `ast.parse(program)` gives us the initial abstract syntax tree. From here, we perform many transformations on this tree in the compiler pipeline. These transformations are centered around moving towards a flattened intermediate representation that can be easily transformed into assembly language.

3.2 String Data Type

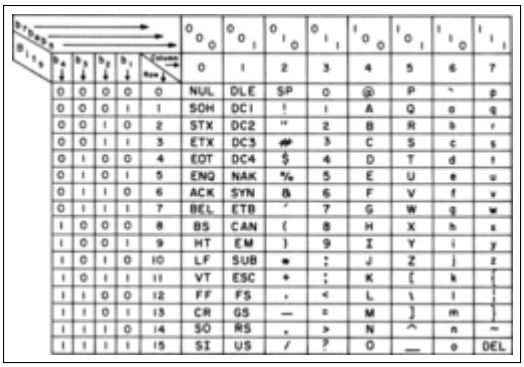
To create the string data type, we used the previous implementation of lists with some simple changes. Because we didn't take the char approach, subscription of a single element will return a single element string.

Flattening a string definition is achieved by creating a string of the proper length and then using `set_subscript()` to populate the string. This is identical to the process for flattening on a list assign.

In explication, string constants are turned into String nodes that contain lists of `InjectInt(Constant())` nodes (where the constant's value is the ASCII number for the character).

3.3 Runtime

In order to support a new datatype, the runtime functions had to be adjusted. This included:



	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	|
13	CR	GS	-	=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

Figure 3. ASCII Number to Character Conversion Table

```
>>> s = "abcdef"
>>> l = [1,2,s,True]
>>> print(s)
abcdef
>>> print(l)
[1, 2, 'abcdef', True]
>>>
```

Figure 4. Python String Printing Inside and Outside of a List

1. Adding a new entity to the `big_type_tag` enumeration.
2. Adding the `string_struct` structure.
3. Updating the `pyobj_struct`.
4. Adding the `create_string()` function that returns a pointer to a new `big_pyobj` of specified length.
5. Adding the `print_string()` function, as well as a call to it in `print_pyobj()`. This function has adaptability to understand if it is printing just a string, or a list item that has to be a string. In the latter case, the string will have quotes around it, matching Python's convention. Much of the syntax of `print_string()` is adapted from `print_list()`. The behavior of string printing is shown in Figure 4.
6. Adding the `print_char()` function for use by the `print_string()` function.
7. Updating the `add()` function to support concatenation of strings using the `+` operator, as well as the `string_add()` helper function. That function is basically identical to the `list_add()` function.
8. Updating the `equal()`, `is_true()`, `subscript()`, and `get_subscript()` functions to include support for strings, which included minor adaptations to the list cases of the functions.

```
>>> original_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> new_iterable = [25, 26, 27, 28, 29]
>>> original_list[::2] = new_iterable
>>> print(original_list)
[25, 1, 26, 3, 27, 5, 28, 7, 29, 9]
```

Figure 5. Python List Slice Assignment

9. Adding the `string_to_big()` function which gets called by `create_string()` and `add()` (for concatenating). This function just tags the big pyobj with the string tag and attaches the string information.
10. Adding the `is_string()` function to check that a pyobj is a big and if so, has the big tag.

To print, we duplicated the behavior of list printing, but also utilized a new function called `print_char()` that takes an unboxed int assumed to be in ascii form.

3.4 Slicing

In order to properly implement string slicing, `get_subscript()` would have to be adjusted. To handle a range of values, `end` and `step` were added as parameters. It will return a newly created string based on the start, end, and step values. It calculates the absolute indexes for the new string, and the size is calculated using the equation:

$$\text{size} = 1 + \frac{|\text{end} - \text{start}| - 1}{|\text{step}|}$$

After the new string is initialized, it is populated using `set_subscript()` by a traversal from start to end by step size (start < end for step > 0).

4 Slice Assignment Implementation for Lists

The grammar for list slice assignment is covered by the string grammar, so no updates to the grammar were necessary for this feature. Most of the logic that handled slicing covered list slice assignment, but there were still a few changes that had to be made. An example of slice assignment is shown in 5.

4.1 Runtime

In order to support list slice assignment, the runtime had to be adjusted. The function that required the most attention was `subscript_assign()`. To include support for slicing, it would need new end and step parameters. If a step was given, we follow this process:

1. Assert size of target section and source list are the same.
2. Update original array in slice range with the new values, inserting instead of retrieving.
3. Return concat of beginning, new middle, and end.

If no step was given, it just returns concat of beginning, new middle, and end. On top of the major adjustments to be made in the function `subscript_assign()`, we needed to:

1. Add the `is_negative()` function that returns a boxed integer (0 or 1) that depends on negativity. This function is used to determine if the specified step size is negative or positive.
2. Add the `get_length()` function that returns the length of a string or list. This a requirement for the implementation of list slice assignment.
3. Add an (redundant) error checking case for the attempt to mutate a string.

Contributions

This project was adapted from the class codebase of Will and Matt, so they focused more on the parts of the project that were written in there. Tucker and Collin were able to put more work into the runtime as it did not require as much prior knowledge of the specific compiler implementation, and Collin focused on making a suite of testcases. That does not include all the work done from everyone as we were working together on most parts but that highlights the focus of each persons work. Everyone contributed to the paper, but Tucker was able to work on most of the sections. Overall we found a productive separation of work.

Conclusion

The implementation of strings in our segment of Python turned out to be an interesting activity, especially learning about the Python-specific behavior of strings (such as equality with other types), and the differences in treatment between strings and list. Also, while editing the runtime functions, there was some interaction with C strings, and thus more interesting behavior was discovered, allowing the comparing and contrasting of Python and C strings. The list slice assignment implementation also proved to be an interest activity, and more challenging than strings, due to the lack of example code to work off of, and, once again, the unique and somewhat infuriating behavior that slicing and stepping exhibit in Python, as seen in Figure 6. This project proved to be a satisfying and logical next step in our growing compiler, as it could enable future implementation of file I/O, and more complex objects, as well as removing some of the reliance on the C runtime functions.

```
>>> l = [1,2,3,4,5,6]
>>> l[::]
[1, 2, 3, 4, 5, 6]
>>> l[::-1]
[6, 5, 4, 3, 2, 1]
>>> l[1:4]
[2, 3, 4]
>>> l[5:1:-2]
[6, 4]
```

Figure 6. Python Slicing Behavior

References

[1] [n. d.]. The COMIT II Manual. http://www.catb.org/~esr/comit/comit-manual.html

[2] [n. d.]. The SNOBOL Programming Language. http://groups.umd.umich.edu/cis/course.des/cis400/snobol/snobol.html

[3] R. J. W. Housden. 1975. On string concepts and their implementation. *Comput. J.* 18 (March 1975), 150–156. https://academic.oup.com/comjnl/article/18/2/150/374127

[4] Ronald Morrison. 1982. The String as a Simple Data Type. *ACM SIGPLAN Notices* 17, 3 (March 1982). https://doi.org/10.1145/947912.947914

[5] Jean E. Sammet. 1972. Programming Languages:History and Future. *Commun. ACM* 15, 7 (July 1972). https://redirect.cs.umbc.edu/courses/undergraduate/331/resources/papers/sammet1972.pdf

[6] Jeremy G. Siek and Bor-Yuh Evan Chang. 2017. A Problem Course in Compilation: From Python to x86 Assembly. (August 2017). Revised by Joseph Izraelevitz in Febuary 2023.