Python String Slicing

K. S. Ooi
Foundation in Science
Faculty of Health and Life Sciences
INTI International University
Persiaran Perdana BBN, Putra Nilai,
71800 Nilai, Negeri Sembilan, Malaysia
E-mail: kuansan.ooi@newinti.edu.my

Abstract

Python slice syntax is nothing new. We find such syntax in other programming languages. However, string slicing in Python is usually covered briefly in standard references, Python textbooks as well as favorite websites. In this article, the author attempts to explore every nook and cranny of string slicing in Python. Besides proving a string slicing theorem, the author explores the limitation of string slicing expressed in everyday languages, the default values of all three arguments of the slice, the minimal string slice statement, the meaning of negative step, the influence of step on other two arguments, and two suggestions to correctly predict the substring of a slice.

Keywords: Python 3, string slicing, slice indices, string slicing theorem.
Date: Dec 25, 2020

1. Introduction

All general-purpose programming languages deal with strings. Modern languages such as Python have tremendous ability to identify text patterns in strings through their implementation of regular expression engine. According to the history, Perl is perhaps the first language that pushes the use of regular expressions to the mainstream. Python support this Perl flavor [1]. The first chapter of Hellman's recent book [2] illustrates the fact string manipulation is dominated by regular expressions. However, this article deals with another string manipulation usually overlooked by Python programmers, that is string splicing. One of the favorite textbooks on Python programming [3] has about one page on string slice. One of the favorite Python learning websites, where beginners turn to for learning Python, W3Schools [4], has less than half a page on slicing string using Python. String splicing, in my opinion, is the prerequisite of regular expressions. I have recently written about string splicing in a tiny section of a book chapter [5]. That section needs revision and expansion, with additional new materials and examples, and correction of some misconceptions I had.

2. String Splicing Theorem

String splicing is about using splice syntax to obtain a substring from a string. It is a rather simple thing to do. Let us have a string object called a_string. The slice syntax is simply given by

(1) a_string[*start*:*end*:*step*]

We view a string as a sequence of characters. In this article, I use the term argument loosely. In equation (1), I will say string slicing has three arguments, *start*, *end*, and *step*. The first character is indexed by 0. And if the length of the string is *n*, the last character is indexed by *n – 1*. So, the range of indices of the string is given by

(2) $i = 0, 1, \ldots, n - 1$ where n = len(a_string)

The *start* index is the *index* of the first character of the substring. However, the *end* index of the string will not be included in the substring. Do not be alarmed. This is how indexing works in Python. For example, *range(2,6)* will return a list of numbers [2, 3, 4, 5], that is the index starts with 2 but not included 6 in the list. The *step* argument defines how you want the characters to jump in the substring, including reversed jumping. I will focus on the *step* argument near the end of this article.

This seems like an innocuous affair until you read Python code littering all over places, which obviously produced with pride. Programmers are people [6]; therefore, we should not be too surprise to find code that is cool but hard to understand. The string slice from programmers is deliberately obtuse. The complication comes from the syntax of slice, which has *sign*. The positive index is the forward indexing; the negative is the reversed indexing. If you start from the first character, the forward index is 0; if you use the reversed index, you start at the last character, which is -1. So, mathematically for a_string, which has length *n*,

(3) a_string[i] = a_string[$i - n$], where $i = 0, 1, \ldots, n - 1$

Let us consider this is as a theorem. Since the first index is 0 and the last index is -1, counting backward, you reach the first character by -*n*.

(4) a_string[0] = a_string[-n]

Similarly, counting forward you reach the last character by counting $n - 1$, and since the reversed index for the last character is -1,

(5) a_string[$n - 1$] = a_string[-1]

By mathematical induction, if the forward index advances on step, the reversed index becomes larger one step as well.

(6) a_string[$i + 1$] = a_string[$i - n + 1$]

And with that we have proved the theorem (3).

With the theorem proving out of the way, we can show one example. Let us have "Jumping-Fox" as our string. The forward and reversed indices are shown in the following figure.

0	1	2	3	4	5	6	7	8	9	10
J	**u**	**m**	**p**	**i**	**n**	**g**	**-**	**F**	**o**	**x**
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Figure 1: Two indexing systems for the "Jumping-Fox" string

The following Python program will print True 11 times on the screen.

Program 1

```
a_string = "Jumping-Fox"
n = len(a_string)
for i in range(n):
    print(a_string[i] == a_string[i-n])
```

3. The Default Values

The equation (1) shows that slicing has three arguments: *start*, *end*, and *step*. The first two arguments are essential, and the step argument is optional. You can leave the arguments blank. Since *start* and *end* are essential, even though can be blank, you have to supply them in slicing. So, the following program will fail to run, because of syntax error, for you fail to supply the arguments of *start* and *end*. So, you cannot use [] in slicing a string.

Program 2 ✗

```
a_string = "Jumping-Fox"
print(a_string[])
```

However, if you supply the two arguments *start* and *end*, it will work, even though you leave them blank, as shown in the following program. The program will print out Jumping-Fox on the screen.

Program 3

```
a_string = "Jumping-Fox"
print(a_string[:])
```

When you leave *start* and *end* blank, in a_string[:], Python will supply the default values. The default values of the three arguments are tabulated below.

Argument	Default value
start	0 or -*n*
end	*n*
step	1

Table 1: Default values of *start*, *end*, and *step*. Bear in mind this table is valid only when your *step* is not a negative value.

The following program will print Jumping-Fox on the screen 9 times, as all the slicing are the same. Observe how the default values play out.

Program 4

```
a_string = "Jumping-Fox"
n = len(a_string)
print(a_string)
print(a_string[:])
print(a_string[::])
print(a_string[0:])
print(a_string[0:n])
print(a_string[:n])
print(a_string[-n:n])
print(a_string[-n:])
print(a_string[0:n:1])
```

However, print(a_string[:-1]) will print Jumping-Fo on the screen, without the x. Now you know why.

4. Expressing String Slicing in Everyday Language

Using the a_string as an example, I attempt to express string slicing in everyday language. I summarize the result in the following table.

Everyday language	Slice	Substring
Get the first 6 characters	a_string[:6]	`Jumpin`
Get the last 6 characters	a_string[-6:]	`ng-Fox`
Obtain the substring containing all characters except the last 3 characters and also except the first 2 characters	a_string[2:-3]	`mping-`
Obtain the substring containing the last 6 characters except the last 2 characters	a_string[-6:-2]	`ng-F`
Obtain the substring of length 5 starting from the second character	a_string[1:6]	`umpin`

Table 2: Slicing in everyday language. The string a_string = "Jumping-Fox".

Based on these 5 examples in Table 2, let us test how well we are able express slicing in everyday language. So, a_string[:-3] means the whole string except the last three characters. This is correct. However, it is difficult to express a_string[-6:7]. You cannot say obtain a substring of length 13, that contains the last 6 characters. Since ng is printed on the screen, the substring we obtain is the characters when these two indices overlap, as shown in the following figure.

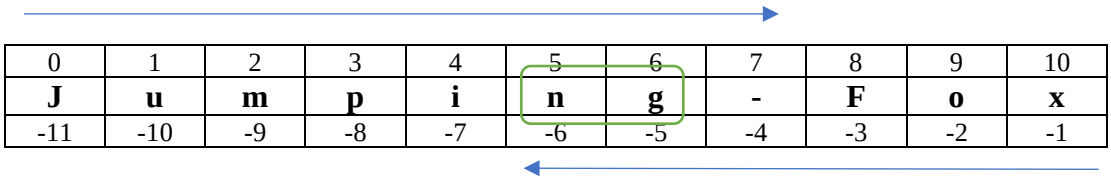

Figure 2: Two indexing systems overlap when you try the slice a_string[-6:7]. Since the substring will not contain the character indexed by 7, we end up getting a substring ng.

It is obvious that we cannot express the entire string slicing into everyday language. We can do it for certain cases, for examples the ones in Table 2. In the end, the overlap of indices of *start* and *end* is the general way to express the slices we get. You may try the overlap of *start* and *end* on the cases of Table 2.

Program 5

```
a_string = "Jumping-Fox"
n = len(a_string)

# Get the first 6 characters
print(a_string[:6])

# Get the last 6 characters
print(a_string[-6:])

# Obtain the substring containing all characters
# except the last 3 characters and
# except the first 2 characters
print(a_string[2:-3])

# Obtain the substring containing the last 6 characters
# except the last 2 characters
print(a_string[-6:-2])

# Obtain the substring of length 5
# starting from the second character
print(a_string[1:6])

# Obtain the substring that contains
# all character except the last 3 characters
```

```
print(a_string[:-3])

# Obtain the substring of length 13
# containing the last 6 characters.
# This doesn't make sense.
print(a_string[-6:7])
```

5. The Step Has Life of Its Own

The *step* can be set either positive or negative. Positive step means forward slicing; negative means reversed slicing. In the following program, the first print statement results in Jumping-Fox, whereas the second one results in xoF-gnipmuJ.

Program 6

```
a_string = "Jumping-Fox"
n = len(a_string)

print(a_string[::+1]) # Jumping-Fox
print(a_string[::-1]) # xoF-gnipmuJ
```

The following program prints `upn-` 4 times.

Program 7

```
a_string = "Jumping-Fox"
n = len(a_string)

print(a_string[1:8:2])
print(a_string[1:-3:2])
print(a_string[-10:8:2])
print(a_string[-10:-3:2])
```

The following program prints `ogp` 4 times.

Program 8

```
a_string = "Jumping-Fox"
n = len(a_string)

print(a_string[9:1:-3])
print(a_string[9:-10:-3])
print(a_string[-2:1:-3])
print(a_string[-2:-10:-3])
```

If you trace the programs 7 and 8, you will have 100% correct substrings if you use

- Theorem 3 to convert all negative *start* and *end* indices to positive values, and interpret the indices as usual. Perform the jump as specified by *step*. If the *step* is negative, reverse the substring and perform the jump.
- Or, you use the overlap indices (such as the example in Table 2) to obtain the substring. Then Perform the jump as specified by *step*. Again, reverse the substring if the *step* is negative and perform the jump.

This is not the end of story. If you set the *step* a negative value, the default of *end* is $-(n + 1)$. This fact is illustrated in the following program, which print the substring `xFgimJ` 4 times.

Program 9

```
a_string = "Jumping-Fox"
n = len(a_string)

print(a_string[::-2])
print(a_string[-1::-2])
print(a_string[10::-2])
print(a_string[10:-12:-2])
```

6. Concluding Remarks

In this article, I have proved the string slicing theorem (3), which can be used as a tool to predict the substring, particularly if you encounter negative values for *start* and *end*. The three arguments of slicing have default values. However, if you reverse the string by setting the *step* a negative value, the default value of *end* will be affected; the default *end* will become $-(n + 1)$ instead of n, where n is the length of the string. An attempt is made to express string slicing in everyday language; however, this does not work for all cases. In the end, if you want to trace a slice and correctly predict the substring, the general way to do it is by converting all negative indices of *start* and *end* to positive value using theorem (3), interpret the indices as usual, perform the jump as specified by the *step*, and reverse the substring if the *step* is a negative value. Or, you can also use the overlap indices technique shown in this article to correctly predict the substring of a slice.

References

1. Felix Lopez and victor Romero, *Mastering Python Regular Expressions*, Packt Publishing, UK (2014)

2. Doug Hellmann, *The Python 3 Standard Library by Example*, Pearson Education, Inc, Boston (2017)

3. Allen B. Downey, *Think Python*, O'Reilly Media, Sebastopol (2016)

4. W3Schools, *Python Slice String*, at https://www.w3schools.com/python/gloss_python_string_slice.asp

5. K. S. Ooi, *The Perils of Idiomatic Python*, Chapter 1, Current STEM. Volume 2, Edited by Maurice H. T. Ling, Nova Science Publishers (2019)

6. Gerald M. Weinberg, The Psychology of Computer Programming, Dorset House (1971)