3.5.Strings: string slices, immutability, string functions and methods, string module, Lists as arrays

A string is a **sequence** of characters. We can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an **index**. The index indicates which character in the sequence is required .To print the character we use

```
>>> print letter
a
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print letter
b
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th(“two-eth”) letter.

We can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise the result will be :

```
>>> letter = fruit[1.5]

TypeError: string indices must be integers, not float
```

Len( )

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print last
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

String Slices:

A segment of a string is called a slice. Selecting a slice is similar to selecting a character.

Syntax:

variable [start:stop]

<String_name> [start:stop]

Example

Char	a=	"B	A	N	A	N	A"
Index from Left		0	1	2	3	4	5
Index from Right		-6	-5	-4	-3	-2	-1

>>>print(a[0]) → prints B #Prints B Alone 0th Position

>>>print(a[5]) → prints A #Prints A Alone Last Position

>>>print(a[-4]) → print N #Print From Backwards -4th Position

>>>a[:] → 'BANANA' #Prints All

>>>print(a[1:4]) → print ANA #Print from 1st Position to 4th Position

>>> print(a[1:-2]) → ANA #Prints from 1st position to -3th Position

Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

The reason for the error is that strings are **immutable**, which means we can't change an existing string. The best we can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

String methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word =
word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking upper on the word.

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

'In this example, we invoke find on word and pass the letter we are looking for as a parameter.

Actually, the find method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start.

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2 (not including 2).

String methods &Descriptions

S.No	Method	Syntax	Description	Example
1.	len()	len(String)	Returns the length of the String	len(a) →5
2.	centre()	centre(width,fullchar)	The String will be centred along with the width specified and the charecters will fill the space	a.centre(20,+) → ++++Hello++++
3.	lower()	String.lower()	Converts all upper case into lower case	b.lower() → hello
4.	upper()	String.upper()	Converts all lower case into upper case	a.upper() → HELLO
5.	capitalize()	String.capitalize()	It converts the first letter into capital	b.capitalize() → HELLO
6.	split()	String.split("Char")	splits according to the character which is present inside the function	c.split("+") → 1+2+3+4+5
7.	join()	String1.join(String2)	It concatenates the string with the sequence	a.join(b) → Hello hELLO
8.	isalnum()	String.isalnum()	It checks the string is alpha numeric or not. If the string contains 1 or more alphanumeric characters it returns 1, else its returns 0	d="a-b" d.isalnum() → returns 1

9.	isalpha()	String.isalpha()	Returns true if it has at least 1 or more alphabet characters, else it return false	b.isalpha() →returns 1
10.	isdigit()	String.isdigit()	Returns true if it has at least 1 or more digits, else it return false	b.isdigit() →returns 0
11.	islower()	String.islower()	Returns true if the string has at least 1 or more Lower case characters, else it return false	b.islower() → returns 1
12.	isupper()	String.isupper()	Returns true if the string has at least 1 or more Upper case characters, else it return false	b.isupper()→ returns 1
13.	isnumeric()	String.isnumeric()	Returns true if the string contains only numeric character or false otherwise	a.isnumeric()→ returns 0
14.	isspace()	String.isspace()	Returns true if the string contains only wide space character or false otherwise	a.isspace() → returns 0 e=”a b” ; e.isspace() → returns 1
15.	istitle()	String.istitle()	Returns true if the string is properly titled or false otherwise	d=”Hello How R U” d.istitle() →returns 1
16.	isdecimal()	String.isdecimal()	Returns true if the string contains decimal value or false otherwise	c.isdecimal() → returns 0
17.	title()	String.title()	Returns title cased, all the characters begin with upper case	d=”hello how h u” d.title() → ”Hello How R U”
18.	find()	String.find(String, start,end)	If the string is found it returns index position or it returns -1	a.find(“He”,0,4) → returns 1 (index position)

19.	endswith()	String.endswith("text" , beg, end)	The string will check for whether the character is ending with the specified character. If it found it returns true, else false	a.endswith(i,0) → returns false
20.	index()	String.index('text',beg , end)	It is same as find(). but it raises exception when the string is not found	a.index('i',0) → returns 1
21.	count()	String.count('text',beg , end)	It counts howmany times a string appears	a.count('i',0) → returns 2
22.	rfind()	String.rfind('text',beg, end)	It finds a string from right to left	a.rfind('i') →-1
23.	rindex()	String.rindex('text',be g, end)	Same as index() but moves from right to left	a.rindex('l',0) → returns 3
24.	rjust()	String.rjust(width,str,fi llchar)	It will justify the character into right and fill with the character	a.rjust(10,a,'-')→ -----Hello
25.	ljust()	String.ljust(width,str,fi llchar)	It will justify the character into left and fill with the character	a.ljust(10,a,'+')→ Hello+++++
26.	rstrip()	rstrip()	It removes all the spaces at the end	rstrip(a) → it returns -1
27.	startswith()	startswith(text, beg, end)	It checks whether the character starts with the specified one	a.stratswith(H,0) → returns true

Strings Modules:

This module contains a number of functions to process standard python strings.

Using import string' we can invoke string functions.

Example: Using the string module

```
import string
text = "Monty Python's Flying Circus"
print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text, "Java")
print "count", "=>", string.count(text, "n")
```

Output:

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Java's Flying Circus
find => 6 -1
count => 3
```

The in **Operator**

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 't' in 'python'
True
>>> 'jan' in 'python'
False
```

For example, the following function prints all the letters from word1 that also appear in word2:

```
def in_both(word1, word2):
for letter in word1:
```

```
if letter in word2:
print(letter)
>>> in_both('django','mongodb')
d
n
g
o
```

List as Array

To store such data, in Python uses the data structure called list (in most programming languages the different term is used — “array”).

Arrays are sequence types and like lists, except that the type of objects stored in them is constrained.

A list (array) is a set of objects.

Individual objects can be accessed using ordered indexes that represent the position of each object within the list (array).

The list can be set manually by enumerating of the elements the list in squarebrackets, like here:

Primes = [2, 3, 5, 7, 11, 13]

Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']

The list Primes has 6 elements, namely: Primes[0] == 2, Primes[1] == 3,

Primes[2] == 5, Primes[3] == 7,Primes[4] == 11, Primes[5] == 13.

The list Rainbow has 7 elements, each of which is the string.

Like the characters in the string, the list elements can also have negative index, for example, Primes[-1] == 13,Primes[-6] == 2.

Several ways of creating and reading lists.

First of all, we can create an empty list and can add items to the end of list using append.

Example

```
a = []                                          # start an empty list
n = int(input('Enter No of Elements'))          # read number of element in the list
for i in range(n):
new_element = int(input('Enter Element :'))     # read next element
a.append(new_element)                           # add it to the list
one:                                            # the last two lines could be replaced by
print(a)                                        # a.append(int(input('Enter Element :')))
```

Result

```
Enter No of Elements5
Enter Element :2
Enter Element :7
Enter Element :4
Enter Element :3
Enter Element :8
[2, 7, 4, 3, 8]
```