CHAPTER 6

Python Strings Revisited

• Concatenating, Appending, Multiplying Strings • String Formatting Operator • Built-in String Methods and Functions • Slice, subscript, in and not in Operators • Comparing and Iterating Strings • The String Module

INTRODUCTION

The Python string data type is a sequence made up of one or more individual characters, where a character could be a letter, digit, whitespace, or any other symbol. Python treats strings as contiguous series of characters delimited by single, double or even triple quotes. Python has a built-in string class named `"str"` that has many useful features. We can simultaneously declare and define a string by creating a variable of string type. This can be done in several ways which are as follows:

```
name = "India"                graduate = 'N'
country = name                nationality = str("Indian")
```

Here, `name`, `graduate`, `country`, and `nationality` are all string variables.

We have already seen in Chapter 3 that string literals can be enclosed by either triple, double or single quotes. Escape sequences work with each type of string literals. A multiple-line text within quotes must have a backslash \ at the end of each line to escape the new line.

Indexing: Individual characters in a string are accessed using the subscript ([ ]) operator. The expression in brackets is called an *index*. The index specifies a member of an ordered set and in this case it specifies the character we want to access from the given set of characters in the string.

The index of the first character is `0` and that of the last character is `n-1` where `n` is the number of characters in the string. If you try to exceed the bounds (below `0` or above `n-1`), then an error is raised.

Traversing a String: A string can be traversed by accessing character(s) from one index to another. For example, the following program uses indexing to traverse a string from the first character to the last.

Example 6.1 Program to demonstrate string traversal using indexing

```
message = "Hello!"
index = 0
for i in message:
  print("message[", index, "] = ", i)
index += 1
```

OUTPUT

```
message[ 0 ] =  H
message[ 1 ] =  e
message[ 2 ] =  l
message[ 3 ] =  l
message[ 4 ] =  o
message[ 5 ] =  !
```

We see that there are 6 characters in the message, if we try to access 7th character by writing, `print(message[7])`, then the `IndexError: string index out of range error` will be generated. Index can either be an integer or an expression that evaluates to an integer.

Example 6.2 Program to demonstrate an expression used as an index of a string

```
str = "Hello, welcome to the world of Python"
i = 2
print(str[i])   # index is an integer
print(str[i*3+1])  # index is an expression that evaluates to an integer
```

OUTPUT

```
l
w
```

Therefore, when you try to execute the following code, an error will be generated.

```
str = "Hello, welcome to the world of Python"
print(str['o'])
TypeError: string indices must be integers, not str
```

Also note that even the whitespace characters, exclamation mark and any other symbol (like `?`, `<`, `>`, `*`, `@`, `#`, `$`, `%`, etc.) that forms a part of the string would be assigned its own index number.

6.1 CONCATENATING, APPENDING, AND MULTIPLYING STRINGS

The word concatenate means to join together. Python allows you to concatenate two strings using the + operator as shown in the following example.

Example 6.3 Program to concatenate two strings using + operator

```
str1 = "Hello "
str2 = "World"
str3 = str1 + str2
print("The concatenated string is : ", str3)
```

OUTPUT

```
The concatenated string is :  Hello World
```

Append mean to add something at the end. In Python you can add one string at the end of another string using the += operator as shown below.

Example 6.4 Program to append a string using += operator

```
str = "Hello, "
name = input("\n Enter your name : ")
str += name
str += ". Welcome to Python Programming."
print(str)
```

OUTPUT

```
Enter your name : Arnav
Hello, Arnav. Welcome to Python Programming.
```

You can use the * operator to repeat a string *n* number of times.

Example 6.5 Program to repeat a string using * operator

```
str = "Hello"
print(str * 3)
```

OUTPUT

```
HelloHelloHello
```

The `str()` function is used to convert values of any other type into string type. This helps the programmer to concatenate a string with any other data which is otherwise not allowed.

```
str1 = "Hello"
var = 7
str2 = str1 + var
print(str2)
```

OUTPUT

```
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 3,
  in <module>
    str2 = str1 + var
TypeError: cannot concatenate 'str'
and 'int' objects
```

```
str1 = "Hello"
var = 7
str2 = str1 + str(var)
print(str2)
```

OUTPUT

```
Hello7
```


The `print` statement prints one or more literals or values in a new line. If you don't want to print on a new line then, add `end` statement with a separator like whitespace, comma, etc. as shown below.

```
print("Hello")
print("World")
```

OUTPUT

```
Hello
World
```

```
print("Hello", end = ' ')
print("World")
```

OUTPUT

```
Hello World
```

A raw string literal which is prefixed by an `'r'` passes all the characters as it is. They are not processed in any special way, not even the escape sequences. Look at the output of the code carefully to understand this difference.

Example 6.6 Program to print a raw string

```
print("\n Hello")
print(r"\n World")
```

OUTPUT

```
Hello
\n World
```

Note The `'u'` prefix is used to write Unicode string literals.

6.2 STRINGS ARE IMMUTABLE

Python strings are immutable which means that once created they cannot be changed. Whenever you try to modify an existing string variable, a new string is created.

Every object in Python is stored in memory. You can find out whether two variables are referring to the same object or not by using the `id()`. The `id()` returns the memory address of that object. As both `str1` and `str2` points to same memory location, they both point to the same object.

Example 6.7 Program to demonstrate string references using the `id()` function

```
str1 = "Hello"
print("Str1 is : ", str1)
print("ID of str1 is : ", id(str1))
str2 = "World"
print("Str2 is : ", str2)
```

```
print("ID of str1 is : ", id(str2))
str1 += str2
print("Str1 after concatenation is : ", str1)
print("ID of str1 is : ", id(str1))
str3 = str1
print("str3 = ", str3)
print("ID of str3 is : ", id(str3))
```

OUTPUT

```
Str1 is :  Hello
ID of str1 is :  45093344
Str2 is :  World
ID of str1 is :  45093312
Str1 after concatenation is :  HelloWorld
ID of str1 is :  43861792
str3 =  HelloWorld
ID of str3 is :  43861792
```

From the output, it is very clear that `str1` and `str2` are two different string objects with different values and have a different memory address. When we concatenate `str1` and `str2`, a new string is created because strings are immutable in nature. You can check this fact by observing the current and previous address of `str1`.

Finally, we create a new string variable `str3` and initialize it with `str1`. Since they both point to the same value, their address is exactly same. Now can you guess the output of the following code:

```
str = "Hi"
str[0] = 'B'
print(str)
```

Yes, the code will result in an error— `TypeError: 'str' object does not support item assignment` simply because strings are immutable. If you want to make any kind of changes you must create a new string as shown below.

```
str = "Hi"
new_str = "Bi"
print("Old String = ",str)
print("New String = ",new_str)
```

OUTPUT

```
Old String =  Hi
New String =  Bi
```

Note We cannot delete or remove characters from a string. However, we can delete the entire string using the keyword `del`.

6.3 STRING FORMATTING OPERATOR

If you are a C programmer, then you are already familiar with % sign. This string formatting operator is one of the exciting features of Python. The % operator takes a format string on the left (that has %d, %s, etc.) and the corresponding values in a tuple (will be discussed in subsequent chapter) on the right. The format operator, % allows users to construct strings, replacing parts of the strings with the data stored in variables. The syntax for the string formatting operation is:

Programming Tip: A tuple is made of values separated by commas inside parentheses.

```
"<FORMAT>" % (<VALUES>)
```

The statement begins with a *format* string consisting of a sequence of characters and *conversion specifications*. Conversion specifications start with a % operator and can appear anywhere within the string. Following the format string is a % sign and then a set of values, one per conversion specification, separated by commas and enclosed in parenthesis. If there is a single value then parenthesis is optional. Just observe the code given below carefully.

Example 6.8 Program to use format sequences while printing a string

```
name = "Aarish"
age = 8
print("Name = %s and Age = %d" %(name, age))
print("Name = %s and Age = %d" %("Anika", 6))
```

OUTPUT

```
Name = Aarish and Age = 8
Name = Anika and Age = 6
```

In the output we can see that %s has been replaced by a string and %d has been replaced by an integer value. The values to be substituted are provided at the end of the line—in brackets prefixed by %. You can either supply these values directly or by using variables. Table 6.1 lists other string formatting characters.

Note that the number and type of values in the tuple should match the number and type of format sequences or conversion specifications in the string, otherwise an error is returned.

```
>>> '%d %f %s' % (100, 23.89)
TypeError: not enough arguments for format string
Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module>
 "%f" %"abc"
TypeError: float argument required, not str
```

In the first case, number of arguments don't match and in the second case the type of argument didn't match. Hence, the error.

The following Table 6.1 lists some format characters used for printing different types of data.

Table 6.1 Formatting Symbols

Format Symbol	Purpose
%c	Character
%d or %i	Signed decimal integer
%s	String
%u	Unsigned decimal integer

(Contd)

Table 6.1 (*Contd*)

Format Symbol	Purpose
%o	Octal integer
%x or %X	Hexadecimal integer (x for lower case characters a–f and X for upper case characters A–F)
%e or %E	Exponential notation
%f	Floating point number
%g or %G	Short numbers in floating point or exponential notation

To further understand the power of formatting strings, execute the code given below and observe the output, how weird and unorganized it looks.

Example 6.9 Program to display powers of a number without using formatting characters

```
i = 1
print("i\ti**2\ti**3\ti**4\ti**5\ti**6\ti**7\ti**8\ti**9\ti**10")
while i <= 10:
    print(i, '\t', i**2, '\t', i**3, '\t', i**5, '\t', i**10, '\t', i**20)
    i += 1
```

OUTPUT

```
i       i**2    i**3    i**4    i**5            i**6    i**7    i**8    i**9    i**10
1       1       1       1       1               1
2       4       8       32      1024            1048576
3       9       27      243     59049           3486784401
4       16      64      1024    1048576         1099511627776
5       25      125     3125    9765625         95367431640625
6       36      216     7776    60466176        3656158440062976
7       49      343     16807   282475249       79792266297612001
8       64      512     32768   1073741824      1152921504606846976
9       81      729     59049   3486784401      12157665459056928801
10      100     1000    100000  10000000000     100000000000000000000
```

The program prints a table that prints powers of numbers from 1 to 10. Tabs are used to align the columns of values. We see that as the digits increases the columns becomes misaligned. Let's look at a different version of this program which gives a very clean and clear output.

Example 6.10 Program to display powers of a number using formatting characters

```
i = 1
print("%-4s%-5s%-6s%-8s%-13s%-15s%-17s%-19s%-21s%-23s" % \
    ('i', 'i**2', 'i**3', 'i**4', 'i**5', 'i**6', 'i**7', 'i**8', 'i**9', 'i**10'))
while i <= 10:
print("%-4d%-5d%-6d%-8d%-13d%-15d%-17d%-19d%-21d%-23d" % (i, i**2, i**3, i**4,
i**5, i**6, i**7, i**8, i**9, i**10))
    i += 1
```


OUTPUT

```
i   i**2  i**3  i**4  i**5  i**6   i**7    i**8     i**9      i**10
1   1     1     1     1     1     1       1        1         1
2   4     8     16    32    64    128     256      512       1024
3   9     27    81    243   729   2187    6561     19683     59049
4   16    64    256   1024  4096  16384   65536    262144    1048576
5   25    125   625   3125  15625 78125   390625   1953125   9765625
6   36    216   1296  7776  46656 279936  1679616  10077696  60466176
7   49    343   2401  16807 117649 823543 5764801  40353607  282475249
8   64    512   4096  32768 262144 2097152 16777216 134217728 1073741824
9   81    729   6561  59049 531441 4782969 43046721 387420489 3486784401
10  100   1000  10000 100000 1000000 10000000 100000000 1000000000 10000000000
```

In the above code, we have set the width of each column independently using the string formatting feature of Python. The - after each % in the conversion string indicates left justification. The numerical values specify the minimum length. Therefore, %-15d means it is a left justified number that is at least 15 characters wide.

Note You don't need to type semi-colon at the end of each line in Python because Python treats each line of code as a separate statement.

6.4 BUILT-IN STRING METHODS AND FUNCTIONS

Strings are an example of Python *objects*. As discussed earlier, an object is an entity that contains both data (the actual string itself) as well as functions to manipulate that data. These functions are available to any *instance* (variable) of the object.

Python supports many built-in methods to manipulate strings. A method is just like a function. The only difference between a function and method is that a method is invoked or called on an object. For example, if the variable str is a string, then you can call the upper() method as str.upper() to convert all the characters of str in uppercase. Table 6.2 discusses some of the most commonly used string methods.

Table 6.2 Commonly Used String Methods

Function	Usage	Example
capitalize()	This function is used to capitalize first letter of the string.	str = "hello" print(str.capitalize()) **OUTPUT** Hello
center(width, fillchar)	Returns a string with the original string centered to a total of width columns and filled with fillchar in columns that do not have characters.	str = "hello" print(str.center(10, '*')) **OUTPUT** **hello***

(*Contd*)

Table 6.2 (*Contd*)

Function	Usage	Example

`count(str, beg, end)`	Counts number of times str occurs in a string. You can specify beg as 0 and end as the length of the message to search the entire string or use any other value to just search a part of the string.	`str = "he"`
		`message = "helloworldhellohello"`
		`print(message. count (str,0, len (message)))`
		OUTPUT
		`3`

`endswith (suffix, beg, end)`	Checks if string ends with suffix; returns `True` if so and `False` otherwise. You can either set `beg = 0` and end equal to the length of the message to search entire string or use any other value to search a part of it.	`message = "She is my best friend"`
		`print(message.endswith("end", 0,len(message)))`
		OUTPUT
		`True`

`startswith (prefix, beg, end)`	Checks if string starts with prefix; if so, it returns `True` and `False` otherwise. You can either set `beg = 0` and end equal to the length of the message to search entire string or use any other value to search a part of it.	`str = "The world is beautiful"`
		`print(str.startswith ("Th",0, len(str)))`
		OUTPUT
		`True`

`find(str, beg, end)`	Checks if `str` is present in string. If found it returns the position at which `str` occurs in string, otherwise returns `-1`. You can either set `beg = 0` and end equal to the length of the message to search entire string or use any other value to search a part of it.	`message = "She is my best friend"`
		`print(message. find("my",0, len (message)))`
		OUTPUT
		`7`

`index(str, beg, end)`	Same as find but raises an exception if `str` is not found.	`message = "She is my best friend"`
		`print(message.index("mine", 0, len(message)))`
		OUTPUT
		`ValueError: substring not found`

`rfind(str, beg, end)`	Same as find but starts searching from the end.	`str = "Is this your bag?"`
		`print(str.rfind("is", 0, len(str)))`
		OUTPUT
		`5`

`rindex(str, beg, end)`	Same as `rindex` but start searching from the end and raises an exception if `str` is not found.	`str = "Is this your bag?"`
		`print(str.rindex("you", 0, len(str)))`
		OUTPUT
		`8`

(*Contd*)

Table 6.2 (*Contd*)

Function	Usage	Example

`isalnum()`	Returns True if string has at least 1 character and every character is either a number or an alphabet and False otherwise.	`message = "JamesBond007"`
		`print(message.isalnum())`
		OUTPUT
		`True`

`isalpha()`	Returns True if string has at least 1 character and every character is an alphabet and False otherwise.	`message = "JamesBond007"`
		`print(message.isalpha())`
		OUTPUT
		`False`

`isdigit()`	Returns True if string contains only digits and False otherwise.	`message = "007"`
		`print(message.isdigit())`
		OUTPUT
		`True`

`islower()`	Returns True if string has at least 1 character and every character is a lowercase alphabet and False otherwise.	`message = "Hello"`
		`print(message.islower())`
		OUTPUT
		`False`

`isspace()`	Returns True if string contains only whitespace characters and False otherwise.	`message = "  "`
		`print(message.isspace())`
		OUTPUT
		`True`

`isupper()`	Returns True if string has at least 1 character and every character is an upper case alphabet and False otherwise.	`message = "HELLO"`
		`print(message.isupper())`
		OUTPUT
		`True`

`len(string)`	Returns the length of the string.	`str = "Hello"`
		`print(len(str))`
		OUTPUT
		`5`

`ljust(width[, fillchar])`	Returns a string left-justified to a total of `width` columns. Columns without characters are padded with the character specified in the `fillchar` argument.	`str = "Hello"`
		`print(str.ljust(10, '*'))`
		OUTPUT
		`Hello*****`

`rjust(width[, fillchar])`	Returns a string right-justified to a total of `width` columns. Columns without characters are padded with the character specified in the `fillchar` argument.	`str = "Hello"`
		`print(str.rjust(10, '*'))`
		OUTPUT
		`*****Hello`

(*Contd*)

Table 6.2 (*Contd*)

Function	Usage	Example

`zfill (width)`	Returns string left padded with zeros to a total of width characters. It is used with numbers and also retains its sign (+ or -).	`str = "1234"`
		`print(str.zfill(10))`
		OUTPUT
		`0000001234`

`lower()`	Converts all characters in the string into lowercase.	`str = "Hello"`
		`print(str.lower())`
		OUTPUT
		Hello

`upper()`	Converts all characters in the string into uppercase.	`str = "Hello"`
		`print(str.upper())`
		OUTPUT
		`HELLO`

`lstrip()`	Removes all leading whitespace in string.	`str = "Hello"`
		`print(str.lstrip())`
		OUTPUT
		Hello

`rstrip()`	Removes all trailing whitespace in string.	`str = "  Hello  "`
		`print(str.lstrip())`
		OUTPUT
		Hello

`strip()`	Removes all leading and trailing whitespace in string.	`str = "Hello "`
		`print(str.strip())`
		OUTPUT
		Hello

`max(str)`	Returns the highest alphabetical character (having highest ASCII value) from the string `str`.	`str = "hello friendz"`
		`print(max(str))`
		OUTPUT
		Z

`min(str)`	Returns the lowest alphabetical character (lowest ASCII value) from the string `str`.	`str = "hello friendz"`
		`print(min(str))`
		OUTPUT
		D

`replace(old, new [, max])`	Replaces all or max (if given) occurrences of old in string with new.	`str = "hello hello hello"`
		`print(str.replace("he", "FO"))`
		OUTPUT
		`FOlloFOlloFOllo`

Programming Tip: The empty parentheses means that this method takes no argument.

(*Contd*)

Table 6.2 (*Contd*)

Function	Usage	Example

`title()`	Returns string in title case.	`str = "The world is beautiful"`
		`print(str.title())`
		OUTPUT
		`The World Is Beautiful`

`swapcase()`	Toggles the case of every character (uppercase character becomes lowercase and vice versa).	`str = "The World Is Beautiful"`
		`print(str.swapcase())`
		OUTPUT
		`tHE wORLD iS bEAUTIFUL`

`split(delim)`	Returns a list of substrings separated by the specified delimiter. If no delimiter is specified then by default it splits strings on all whitespace characters.	`str = "abc,def, ghi,jkl"`
		`print(str.split(','))`
		OUTPUT
		`['abc', 'def', ' ghi', 'jkl']`

`join(list)`	It is just the opposite of split. The function joins a list of strings using the delimiter with which the function is invoked.	`print('-'.join(['abc', 'def', 'ghi', 'jkl']))`
		OUTPUT
		`abc-def-ghi-jkl`

`isidentifier()`	Returns True if the string is a valid identifier.	`str = "Hello"`
		`print(str.isidentifier())`
		OUTPUT
		`True`

`enumerate(str)`	Returns an enumerate object that lists the index and value of all the characters in the string as pairs.	`str = "Hello WOrld"`
		`print(list(enumerate(str)))`
		OUTPUT
		`[(0, 'H'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o'), (5, ' '), (6, 'W'), (7, 'O'), (8, 'r'), (9, 'l'), (10, 'd')]`

Note that the `strip()` when used with a string argument will strip (from both ends) any combination of the specified characters in the string as shown below.

Example 6.11 To demonstrate strip method on a string object

```
str = "abcdcbabcdcbabcdabcdbcabcdcba"
print(str.strip('abc'))
```

OUTPUT

```
dcbabcdcbabcdabcdbcabcd
```

Let us discuss two more important functions—`format()` and `splitlines()`.

- The `format()` function used with strings is a very versatile and powerful function used for formatting strings. Format strings have curly braces {} as placeholders or replacement fields which gets replaced. We can even use positional arguments or keyword arguments to specify the order of fields that have to be replaced. Consider the code given in the following example and carefully observe the sequence of fields in the output and then compare the sequence as given in the arguments of the `format()` function.

Example 6.12 Program to demonstrate `format()` function

```
str1 = "{}, {} and {}".format('Sun','Moon','Stars')
print("\n The default sequence of arguments is : " + str1)
str2 = "{1}, {0} and {2}".format('Sun','Moon','Stars')
print("\n The positional sequence of arguments (1, 0 and 2) is : " + str2)
str3 = "{c}, {b} and {a}".format(a='Sun',b='Moon',c='Stars')
print("\n The keyword sequence of arguments is : " + str3)
```

OUTPUT

```
The default sequence of arguments is : Sun, Moon and Stars
The positional sequence of arguments (1, 0 and 2) is : Moon, Sun and Stars
The keyword sequence of arguments is : Stars, Moon and Sun
```

- The `splitlines()` returns a list of the lines in the string. This method uses the newline characters like `\r` or `\n` to split lines. Line breaks are not included in the resulting list unless `keepends` is given as True. The syntax of `splitlines()` is given as,

```
str.splitlines([keepends])
```

In the syntax, `keepends` is optional. Look at the code given below and observe its output, both when `keepends` is specified as `True` and when `keepends` is not specified at all.

```
print('Sun and \n\n Stars, Planets \r and Moon\r\n'.splitlines())
print('Sun and \n\n Stars, Planets \r and Moon\r\n'.splitlines(True))
```

OUTPUT

```
['Sun and ', '', ' Stars, Planets ', ' and Moon']
['Sun and \n', '\n', ' Stars, Planets \r', ' and Moon\r\n']
```

Note The `isX` string methods are used to validate user input.

6.5 SLICE OPERATION

A substring of a string is called a *slice*. The slice operation is used to refer to sub-parts of sequences (we will read about them in subsequent chapters) and strings. You can take subset of a string from the original string by using `[]` operator also known as *slicing operator*. Before reading about the slice operation, let us first consider the index of characters in a string as shown in Figure 6.1.

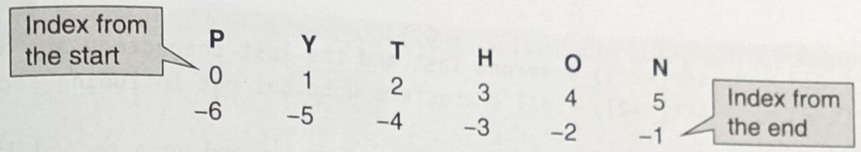

Figure 6.1 Indices in a string

The syntax of slice operation is `s[start:end]`, where `start` specifies the beginning index of the substring and `end-1` is the index of the last character. Now let us take an example, if we have a string `str = "PYTHON"` then the index of characters starting from first character and from the last character can be given as shown in the Figure 6.1. Look at the slice operations given below and observe the output on vis-a-vis our string.

Programming Tip: Calling a method is also known as method invocation.

Note Omitting either start or end index by default takes start or end of the string. Omitting both means the entire string.

Example 6.13 Program to demonstrate slice operation on string objects

```
str = "PYTHON"
print("str[1:5] = ", str[1:5])   #characters starting at index 1 and extending up
to but not including index 5
print("str[:6] = ", str[:6])     # defaults to the start of the string
print("str[1:] = ", str[1:])      # defaults to the end of the string
print("str[:] = ", str[:])        # defaults to the entire string
print("str[1:20] = ", str[1:20])  # an index that is too big is truncated down to
length of the string
```

OUTPUT

```
str[1:5] =  YTHO
str[:6] =  PYTHON
str[1:] =  YTHON
str[:] =  PYTHON
str[1:20] =  YTHON
```

Programming Tip: Python does not have any separate data type for characters. They are represented as a single character string.

Python gives you the flexibility to either access a string from the first character or from the last character. If we access the string from the first character then we use a zero based index, but when doing it backward the index starts with `-1`.

Note Python uses negative numbers to access the characters at the end of the string. Negative index numbers count back from the end of the string.

Example 6.14 Program to understand how characters in a string are accessed using negative indexes

```
str = "PYTHON"
print("str[-1] = ", str[-1])   # last character is accessed
print("str[-6] = ", str[-6])   # first character is accessed
```

```
print("str[-2:] = ", str[-2:]) # second last and the last characters are accessed
print("str[:-2] = ", str[:-2]) # all characters upto but not including second last
character
print("str[-5:-2] = ", str[-5:-2]) # characters from second upto second last are
accessed
```

OUTPUT

```
str[-1] =  N
str[-6] =  P
str[-2:] =  ON
str[:-2] =  PYTH
str[-5:-2] =  YTH
```

By observing the outputs of the two codes mentioned in Examples 6.13 and 6.14, we can draw following inferences:

- Elements are accessed from left towards right.
- For any index n, `s[:n] + s[n:] = s`. This is true even if n is a negative number. So, we can say that the slice operation `s[:n]` and `s[n:]` always partition the string into two parts such that all characters are conserved.

Note When using negative index numbers, start with the lower number first as it occurs earlier in the string.

6.5.1 Specifying Stride While Slicing Strings

In the slice operation, you can specify a third argument as the `stride`, which refers to the number of characters to move forward after the first character is retrieved from the string. The default value of stride is `1`. Hence, in all the above examples where the value of stride is not specified, its default value of `1` is used which means that every character between two index numbers is retrieved. The code given below illustrates this difference.

Example 6.15 Program to use slice operation with `stride`

```
str = "Welcome to the world of Python"
print("str[2:10] = ", str[2:10])    # default stride is 1
print("str[2:10:1] = ", str[2:10:1])   # same as stride = 1
print("str[2:10:2] = ", str[2:10:2])   # skips every alternate character
print("str[2:13:4] = ", str[2:13:4])   # skips every fourth character
```

OUTPUT

```
str[2:10] = lcome to
str[2:10:1] = lcome to
str[2:10:2] = loet
str[2:13:4] = le
```

Note that even the whitespace characters are skipped as they are also a part of the string. If you omit the first two arguments and only specify the third one, then the entire string is used in steps (as given by the third argument). This is shown in the following example.

Example 6.16 Program to demonstrate splice operation with just last (positive) argument

```
str = "Welcome to the world of Python"
print("str[::3] = ", str[::3])
```

OUTPUT

```
str[::3] = WceohwloPh
```

You can also specify a negative value for the third argument. This is especially useful to print the original string in reverse order by setting the value of stride to `-1` as shown below.

Example 6.17 Program to demonstrate splice operation with just last (negative) argument

```
str = "Welcome to the world of Python"
print("str[::-1] = ", str[::-1])
```

OUTPUT

```
str[::-1] = nohtyP fo dlrow eht ot emocleW
```

In this example, we have considered the entire original string, reversed it through the negative stride and with a stride of `-3`, we have skipped every third letter of the reversed string.

Example 6.18 Program to print the string in reverse thereby skipping every third character

```
str = "Welcome to the world of Python"
print("str[::-3]", str[::-3])
```

OUTPUT

```
str[::-3] = nt  r ttml
```

6.6 ord() AND chr() FUNCTIONS

The `ord()` function returns the `ASCII` code of the character and the `chr()` function returns character represented by a `ASCII` number. Consider the following examples.

```
ch = 'R'
print(ord(ch))
```

OUTPUT

```
82
```

```
print(chr(82))
```

OUTPUT

```
R
```

```
print(chr(112))
```

OUTPUT

```
p
```

```
print(ord('p'))
```

OUTPUT

```
112
```

6.7 in AND not in OPERATORS

`in` and `not in` operators can be used with strings to determine whether a string is present in another string. Therefore, the `in` and `not in` operator are also known as membership operators.

```
str1 = "Welcome to the world of Python
!!!"
str2 = "the"
if str2 in str1:
    print("Found")
else:
    print("Not Found")
```

OUTPUT

```
Found
```

```
str1 = "This is a very good book"
str2 = "best"
if str2 not in str1:
    print("The book is very good but it
may not be the best one.")
else:
    print ("It is the best book.")
```

OUTPUT

```
The book is very good but it may not be
the best one.
```

You can also use the `in` and `not in` operators to check whether a character is present is in a word. For example, observe the commands and their outputs given below.

```
>>> 'u' in "stars"
False
```

```
>>> 'v' not in "success"
True
```

```
>>> 'vi' in "victory"
True
```

While using the `in` and `not in` operators, remember that a string is a substring of itself as shown below.

```
>>> "world" in "world"
True
```

```
>>> 'a' in 'a'
True
```

6.8 COMPARING STRINGS

Python allows you to compare strings using relational (or comparison) operators such as `>` , `<` , `<=` , etc. Some of these operators along with their description and usage are given in Table 6.3.

Table 6.3 String Comparison Operators and their Description

Operator	Description	Example
`==`	If two strings are equal, it returns True.	`>>> "AbC" == "AbC"` `True`

`!= or <>`	If two strings are not equal, it returns True.	`>>> "AbC" != "Abc"`
		`True`
		`>>> "abc" <> "ABC"`
		`True`

`>`	If the first string is greater than the second, it returns True.	`>>> "abc" > "Abc"` `True`
`<`	If the second string is greater than the first, it returns True.	`>>> "abC" < "abc"` `True`
`>=`	If the first string is greater than or equal to the second, it returns True.	`>>> "aBC" >= "ABC"` `True`
`<=`	If the second string is greater than or equal to the first, it returns True.	`>>> "ABc" <= "ABc"` `True`

These operators compare the strings by using the lexicographical order i.e. using `ASCII` value of the characters. The `ASCII` values of `A-Z` is `65-90` and `ASCII` code for a-z is `97-122`. This means that book is greater than Book because the `ASCII` value of `'b'` is 98 and `'B'` is 66. Let us try more examples.


```
>>> "TED" == "ted"
False
>>> "True" >= "False"
True
```

```
>>> "talk" > "talks"
False
>>> "like" != "likes"
True
```

```
>>> "Main" < "main"
True
>>> "tend" <= "tent"
True
```

Note String values are ordered using lexicographical (dictionary) ordering. The lexicographical order is similar to the alphabetical order that is used with a dictionary (which is discussed in Chapter 8), except that all the uppercase letters come before all the lowercase letters. For example, 'Arman' is less than 'Ben' as the ASCII value for 'A' is 65, and 'B' is 66.

6.9 ITERATING STRING

String is a sequence type (sequence of characters). You can iterate through the string using `for` loop as shown in the code given below.

Example 6.19 Program to iterate a given string using `for` loop

```
str = "Welcome to Python"
for i in str:
print(i, end=' ')
```

OUTPUT

```
W e l c o m e   t o   P y t h o n
```

In the above code, the `for` loop executes for every character in `str`. The loop starts with the first character and automatically ends when the last character is accessed. You can also iterate through the string using `while` loop by writing the following code.

Example 6.20 Program to iterate a given string using `while` loop

```
message = " Welcome to Python "
index = 0
while index < len(message):
   letter = message[index]
   print(letter, end=' ')
   index += 1
```

OUTPUT

```
W e l c o m e   t o   P y t h o n
```

In the above program the loop traverses the string and displays each letter. The loop condition is `index < len(message)`, so the moment index becomes equal to the length of the string, the condition evaluates to `False`, and the body of the loop is not executed. As we said earlier, index of the last character is `len(message) - 1`.

Another point to observe carefully is that *you can iterate through a string either using an index or by using each character in the string*. For example, both the codes given below perform the same job of copying one string into another using the `for` loop but the way you iterate is different—through character or index of the character.

```
# Uses character to iterate
def copy(str):
    new_str = ''
    for i in str:
        new_str += i
    return new_str

str = input("\n Enter a string : ")
print("\n The copied string is : ",
copy(str))
```

OUTPUT

```
Enter a string : Python
The copied string is :  Python
```

```
# Uses index of character to iterate
def copy(str):
    new_str = ''
    for i in range(len(str)):
        new_str += str[i]
    return new_str

str = input("\n Enter a string : ")
print("\n The copied string is : ",
copy(str))
```

OUTPUT

```
Enter a string : Python
The copied string is :  Python
```

PROGRAMMING EXAMPLES

Program 6.1 Write a program to print the following pattern.

```
A
AB
ABC
ABCD
ABCDE
ABCDEF
for i in range(1,7):
      ch = 'A'
      print()
      for j in range(1,i+1):
         print(ch, end=' ')
         ch = chr(ord(ch)+1)
```

Program 6.2 Write a program that takes user's name and PAN card number as input. Validate the information using isX function and print the details.

```
while(1):
    name = input("\n Enter your name : ")
    if name.isalpha() == False:
        print("Invalid Name, Sorry you cannot proceed.")
        break
    else:
        pan_card_no = input("\n Enter your PAN card number : ")
        if pan_card_no.isalnum() == False:
            print("Invalid PAN card Number, Sorry you cannot proceed.")
            break
    print("Please check, "+name+", your PAN card number is : "+pan_card_no)
    break
```


OUTPUT

```
Enter your name : OM
Enter your PAN card number : ABCDE1234F
Please check, OM, your PAN card number is : ABCDE1234F
```

Program 6.3 Write a program that encrypts a message by adding a key value to every character. (Caesar Cipher)

Hint: Say, if key = 3, then add 3 to every character

```
message = "HelloWorld"
index = 0
while index < len(message):
    letter = message[index]
    print(chr(ord(letter) + 3), end=' ')
    index += 1
```

OUTPUT

```
K h o o r Z r u o g
```

Program 6.4 Write a program that uses `split()` to split a multiline string.

```
letter = '''Dear Students,
I am pleased to inform you that,
there is a workshop on Python in college tomorrow.
Everyone should come and
there will also be a quiz in Python, whosoever wins
will win a Gold Medal.'''

print(letter.split('\n'))
```

OUTPUT

```
['Dear Students,', 'I am pleased to inform you that, ', 'there is a workshop on
Python in college tomorrow.', 'Everyone should come and', 'there will also be a
quiz in Python, whosoever wins', 'will win a Gold Medal.']
```

Program 6.5 Write a program to generate an Abecedarian series.

Hint: Abecedarian refers to a series or list in which the elements appear in alphabetical order

```
str1 = "ABCDEFGH"
str2 = "ate"
for letter in str1:
    print((letter + str2), end=' ')
```

OUTPUT

```
Aate    Bate    Cate    Date    Eate    Fate    Gate    Hate
```

Program 6.6 Write a program that accepts a string from user and redisplays the same string after removing vowels from it.

```
def remove_vowels(s):
    new_str = ""
    for i in s:
        if i in "aeiouAEIOU":
            pass
        else:
            new_str += i
    print("The string without vowels is : ", new_str)
str = input("\n Enter a string : ")
remove_vowels(str)
```

OUTPUT

```
Enter a string : The food is very tasty
The string without vowels is :  Th fd s vry tsty
```

Program 6.7 Write a program that finds whether a given character is present in a string or not. In case it is present it prints the index at which it is present. Do not use built-in find functions to search the character.

```
def find_ch(s, c):
    index = 0
    while(index < len(s)):
        if s[index] == c:
            print(c, "found in string at index : ", index)
            return
        else:
            pass
        index += 1
    print (c, " is not present in the string")

str = input("\n Enter a string : ")
ch = input("\n Enter the character to be searched : ")
find_ch(str, ch)
```

Programming Tip: Index numbers allow us to access specific characters within a string.

OUTPUT

```
Enter a string : God is Great
Enter the character to be searched : r
r found in string at index :  8
```

Program 6.8 Write a program that emulates the rfind function.

```
def rfind_ch(s, c):
    index = len(s)-1
    while index>=0:
        if s[index] == c:
            return index
        index = index - 1
    return -1
str = input("\n Enter a string : ")
ch = input("\n Enter the character to be searched : ")
index = rfind_ch(str, ch)
if index != -1:
    print(ch, " is found at location ", index)
else:
    print(ch, "is not present in the string")
```

Programming Tip: The start and end parameters in find() and rfind() are optional.

OUTPUT

```
Enter a string : Let us study Python
Enter the character to be searched : s
s  is found at location  7
```

Program 6.9 Write a program that counts the occurrences of a character in a string. Do not use built-in count function.

```
def count_ch(s, c):
    count = 0
    for i in s:
        if i == c:
            count += 1
    return count
str = input("\n Enter a string : ")
ch = input("\n Enter the character to be searched : ")
count = count_ch(str, ch)
print("In ", str, ch, " occurs ", count, " times")
```

OUTPUT

```
Enter a string : Lovely Flowers
Enter the character to be searched : e
In  Lovely Flowers e  occurs  2  times
```

Program 6.10 Modify the above program so that it starts counting from the specified location.

```
def count_ch(s, c, beg = 0):
    count = 0
    index = beg
    while index < len(s):
        if s[index] == c:
            count += 1
        index += 1
    return count
str = input("\n Enter a string : ")
```

```
ch = input("\n Enter the character to be searched : ")
count = count_ch(str, ch)
print("In ", str, ch, " occurs ", count, " times from beginning to end")
loc = int(input("\n From which position do you want to start counting : "))
count = count_ch(str, ch, loc)
print("In ", str, ch, " occurs ", count, " times from position", loc, " to end")
```

OUTPUT

```
Enter a string : Good Going
Enter the character to be searched : o
In  Good Going o  occurs  3  times from beginning to end
From which position do you want to start counting : 2
In  Good Going o  occurs  2  times from position 2  to end
```

Program 6.11 Write a program to reverse a string.

```
def reverse(str):
    new_str = ''
    i = len(str)-1
    while i>=0:
        new_str += str[i]
        i -= 1
    return new_str

str = input("\n Enter a string : ")
print("\n The reversed string is : ", reverse(str))
```

OUTPUT

```
Enter a string : Python
The reversed string is :  nohtyP
```

Program 6.12 Write a program to parse an email id to print from which email server it was sent and when.

```
info = 'From priti.rao@gmail.com Sun Oct 16 20:29:16 2016'

start = info.find('@') + 1        # Extract characters after @ symbol
end = info.find(".com") + 4       # Extract till m, find returns index of m.
mailserver = info[start:end]      # Extract characters

start = end + 1                   # Ignore whitespace
end = len(info) - 1               # Extract till last character
date_time = info[start:end]

print("The email has been sent through " + mailserver)
print("It was sent on " + date_time)
```

OUTPUT

```
The email has been sent through gmail.com
It was sent on Sun Oct 16 20:29:16 2016
```


6.10 THE STRING MODULE

The string module consists of a number of useful constants, classes, and functions (some of which are deprecated). These functions are used to manipulate strings.

String constants Some constants defined in the string module are:

`string.ascii_letters`: Combination of `ascii_lowercase` and `ascii_uppercase` constants.
`string.ascii_lowercase`: Refers to all lowercase letters from `a-z`.
`string.ascii_uppercase`: Refers to all uppercase letters, `A-Z`.
`string.digits`: Refers to digits from `0-9`.
`string.hexdigits`: Refers to hexadecimal digits, `0-9`, `a-f`, and `A-F`.
`string.lowercase`: A string that has all the characters that are considered lowercase letters.
`string.octdigits`: Refers to octal digits, `0-7`.
`string.punctuation`: String of `ASCII` characters that are considered to be punctuation characters.
`string.printable`: String of printable characters which includes digits, letters, punctuation, and whitespace.
`string.uppercase`: A string that has all the characters that are considered uppercase letters.
`string.whitespace`: A string that has all characters that are considered whitespace like space, tab, linefeed, return, form-feed, and vertical tab.

Example 6.21 Program that uses different methods (upper, lower, split, join, count, replace, and find) on string object

```
str = "Welcome to the world of Python"
print("Uppercase - ", str.upper())
print("Lowercase - ", str.lower())
print("Split - ", str.split())
print("Join - ", '-'.join(str.split()))
print("Replace - ", str.replace("Python", "Java"))
print("Count of o - ", str.count('o') )
print("Find of - ", str.find("of"))
```

OUTPUT

```
Uppercase -  WELCOME TO THE WORLD OF PYTHON
Lowercase -  welcome to the world of python
Split -  ['Welcome', 'to', 'the', 'world', 'of', 'Python']
Join -  Welcome-to-the-world-of-Python
Replace -  Welcome to the world of Java
Count of o -  5
Find of -  21
```

Programming Tip: A method is called by appending its name to the variable name using the period as a delimiter.

To see the contents of the string module, use the `dir()` with the module name as an argument as shown below.

```
>>> dir(string)
['ChainMap', 'Formatter', 'Template', '_TemplateMetaclass', '__builtins__', '__
cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__
spec__', '_re', '_string', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase',
'capwords', 'digits', 'hexdigits', 'octdigits', 'printable', 'punctuation',
'whitespace']
```

To know the details of a particular item, you can use the type command. The function type() takes as an argument the module name followed by the dot operator and the item name.

Example 6.22 Program that displays the type of an item in the string module

```
import string
print(type(string.digits))
```

OUTPUT

```
<class 'str'>
```

```
import string
print(type(string.ascii_letters))
```

OUTPUT

```
<class 'str'>
```

From the output we can see that the type function returns the type of the item in the string module. Just type the following lines and observe the output.

Example 6.23 Program that displays the type of an item in the string module

```
import string
print(string.digits)
```

OUTPUT

```
0123456789
```

```
import string
print(string.ascii_letters)
```

OUTPUT

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

When you try to print what makes the part of digits in a string in Python, all digits from 0-9 are returned. Same is the case with ascii_letters. You can yourself try the rest of the constants defined in the string module. However, to find the details of a particular function, you can print its documentation using the *docstring* through __doc__ attribute as shown below.

Example 6.24 Program to print the doctstring of an item in string module

```
import string
print(string.__builtins__.__doc__)
```

OUTPUT

```
dict() -> new empty dictionary
dict(mapping) -> new dictionary initialized from a mapping object's
    (key, value) pairs
dict(iterable) -> new dictionary initialized as if via:
    d = {}
    for k, v in iterable:
        d[k] = v
dict(**kwargs) -> new dictionary initialized with the name=value pairs
    in the keyword argument list.  For example:  dict(one=1, two=2)
```

You can even use the help() to print the details of a particular item in the string module as shown below.

Example 6.25 Program using help()

```
str = "Hello"
print(help(str.isalpha))
```

OUTPUT

```
Help on built-in function isalpha:
isalpha(...)
    S.isalpha() -> bool
        Return True if all characters in S are alphabetic
    and there is at least one character in S, False otherwise.
None
```

Programming Tip: Passing '\n' in split() allows us to split the multiline string stored in the string variable.

Working with Constants in String Module

You can use the constants defined in the string module along with the find() function to classify characters. For example, if find(lowercase, ch) returns a value except -1, then it means that ch must be a lowercase character. An alternate way to do the same job is to use the in operator or even the comparison operation. All three ways are shown below.

```
# First Way
import string
print(string.find(string.
lowercase, 'g') != -1)
```

OUTPUT

```
True
```

```
# Second Way
import string
print('g' in string.
lowercase)
```

OUTPUT

```
True
```

```
# Third Way
import string
ch = 'g'
print('a' <= ch <= 'z')
```

OUTPUT

```
True
```

Note Type() shows the type of an object and the dir()shows the available methods.

Another very useful constant defined in the string module is *whitespace*. When you write, print (string.whitespace) then all the characters that moves the cursor ahead without printing anything are displayed. These whitespace characters include space, tab, and newline characters.

You can even use the dir() with a string object or a string variable as shown below.

```
str = "Hello"
print(dir(str))
```

OUTPUT

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__
getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__
mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum',
'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
```

```
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Copying and Pasting Strings with the Pyperclip Module

The pyperclip module in Python has copy() and paste() functions that can send text to and receive text from the computer's clipboard. Copying the output of your program to the clipboard makes it easy to paste it to an email, word processor, or some other software.

However, this module does not come with Python and you need to explicitly install it. Once the module is installed, you can type the following commands in IDLE or at the command prompt.

```
import pyperclip
pyperclip.copy('Welcome to the world of Python !!!')
pyperclip.paste()
'Welcome to the world of Python !!!'
```

Note After copying your Python text, if you copy something outside of your program, then the contents of the clipboard will change and the paste() function will return it.

6.11 REGULAR EXPRESSIONS

Regular expressions are a powerful tool for various kinds of string manipulation. These are basically a special text string that is used for describing a search pattern to extract information from text such as code, files, log, spreadsheets, or even documents.

Regular expressions are a *domain specific language* (DSL) that is present as a library in most of the modern programming languages, besides Python. A *regular expression* is a special sequence of characters that helps to match or find strings in another string. In Python, regular expressions can be accessed using the **re** module which comes as a part of the Standard Library. In this section, we will discuss some important methods in the **re** module.

Programming Tip: An exception re.error is raised if any error occurs while compiling or using regular expressions.

6.11.1 The match() Function

As the name suggest, the `match()` function matches a pattern to a string with optional flags. The syntax of `match()` function is,

```
re.match(pattern, string, flags=0)
```

The function tries to match the pattern (which specifies the regular expression to be matched) with a string (that will be searched for the pattern at the beginning of the string). The flag field is optional. Some values of flags are specified in the Table 6.4. To specify more than one flag, you can use the bitwise OR operator as in `re.I | re.M`. If the `re.match()` function finds a match, it returns the match object and None otherwise.

Table 6.4 Different values of flags

Flag	Description
re.I	Case sensitive matching
re.M	Matches at the end of the line
re.X	Ignores whitespace characters
re.U	Interprets letters according to Unicode character set

Example 6.26 Program to demonstrate the use of `match()` function

```
import re
string = "She sells sea shells on the sea shore"
pattern1 = "sells"
if re.match(pattern1, string):
    print("Match Found")
else:
    print(pattern1, "is not present in the string")
pattern2 = "She"
if re.match(pattern2, string):
    print("Match Found")
else:
    print(pattern2, "is not present in the string")
```

OUTPUT

```
sells is not present in the string
Match Found
```

In the above program, 'sells' is present in the string but still we got the output as match not found. This is because the `re.match()` function finds a match only at the beginning of the string. Since, the word 'sells' is present in the middle of the string, hence the result.

Note On success, match() function returns an object representing the match, else returns None.

6.11.2 The search() Function

In the previous function, we saw that even when the pattern was present in the string, None was returned because the match was done only at the beginning of the string. So, we have another function, i.e. `search()`, in the `re` module that searches for a pattern anywhere in the string. The syntax of the `search()` function can be given as,

Programming Tip: While using regular expressions, always use raw strings.

```
re.search(pattern, string, flags=0)
```

The syntax is similar to the `match()` function. The function searches for first occurrence of *pattern* within a *string* with optional *flags*. If the search is successful, a *match* object is returned and None otherwise.

Example 6.27 Program to demonstrate the use of `search()` function

```
import re
string = "She sells sea shells on the sea shore"
pattern = "sells"
if re.search(pattern, string):
    print("Match Found")
```

```
else:
    print(pattern, "is not present in the string")
```

OUTPUT

```
Match Found
```

Note The `re.search()` finds a match of a pattern anywhere in the string.

6.11.3 The sub() Function

The `sub()` function in the `re` module can be used to search a pattern in the string and replace it with another pattern. The syntax of `sub()` function can be given as,

```
re.sub(pattern, repl, string, max=0)
```

According to the syntax, the `sub()` function replaces all occurrences of the `pattern` in `string` with `repl`, substituting all occurrences unless any `max` value is provided. This method returns a modified string.

Example 6.28 Program to demonstrate the use of `sub()` function

```
import re
string = "She sells sea shells on the sea shore"
pattern = "sea"
repl = "ocean"
new_string = re.sub(pattern, repl, string, 1)
print(new_string)
```

OUTPUT

```
She sells ocean shells on the sea shore
```

In the above program, note that only one occurrence was replaced and not all because we had provided 1 as the value of `max`.

6.11.4 The findall() and finditer() Functions

The `findall()` function is used to search a string and returns a list of matches of the pattern in the string. If no match is found, then the returned list is empty. The syntax of `match()` function can be given as,

```
matchList = re.findall(pattern, input_str, flags=0)
```

Example 6.29 Program to demonstrate the use of `findall()` function

```
import re
pattern = r"[a-zA-Z]+ \d+"
matches = re.findall(pattern, "LXI 2013, VXI 2015, VDI 20104, Maruti Suzuki Cars in
India")
for match in matches:
    print(match, end = " ")
```

OUTPUT

```
LXI 2013  VXI 2015  VDI 20104
```


Note The `re.findall()` function returns a list of all substrings that match a pattern.

In the above code, the regular expression, `pattern = r"[a-zA-Z]+ \d+"`, finds all patterns that begin with one or more characters followed by a space and then followed by one or more digits.

The `finditer()` function is same as `findall()` function but instead of returning match objects, it returns an iterator. This iterator can be used to print the index of match in the given string.

Example 6.30 Program to demonstrate the use of `finditer()` function

```
import re
pattern = r"[a-zA-Z]+ \d+"
matches = re.finditer(pattern, "LXI 2013, VXI 2015, VDI 20104, Maruti Suzuki Cars
availble with us")
for match in matches:
    print("Match found at starting index : ", match.start())
    print("Match found at ending index : ", match.end())
    print("Match found at starting and ending index : ", match.span())
```

OUTPUT

```
Match found at starting index :  0
Match found at ending index :  8
Match found at starting and ending index :  (0, 8)
Match found at starting index :  10
Match found at ending index :  18
Match found at starting and ending index :  (10, 18)
Match found at starting index :  20
Match found at ending index :  29
Match found at starting and ending index :  (20, 29)
```

Note that the `start()` function returns the starting index of the first match in the given string. Similarly, we have `end()` function which returns the ending index of the first match. Another method, `span()` returns the starting and ending index of the first match as a tuple.

Note The match object returned by `search()`, `match()`, and `findall()` functions have `start()` and `end()` methods, that returns the starting and ending index of the first match.

6.11.5 Flag Options

The `search()`, `findall()`, and `match()` functions of the module take options to modify the behavior of the pattern match. Some of these flags are:

`re.I or re.IGNORECASE`—Ignores case of characters, so `"Match", "MATCH", "mAtCh"`, etc are all same

`re.S or re.DOTALL`—Enables dot (.) to match newline character. By default, dot matches any character other than the newline character.

`re.M or re.MULTILINE`—Makes the ^ and $ to match the start and end of each line. That is, it matches even after and before line breaks in the string. By default, ^ and $ matches the start and end of the whole string.

`re.L or re.LOCALE`—Makes the flag `\w` to match all characters that are considered letters in the given current locale settings.

`re.U or re.UNICODE`—Treats all letters from all scripts as word characters.

6.12 METACHARACTERS IN REGULAR EXPRESSION

Metacharacters make regular expressions more powerful than normal string methods. They allow you to create regular expressions to represent concepts like "one or more repetitions of a vowel".

Python allows users to specify metacharacters (like +, ?, ., *, ^, $, (), [], {}, |, \) in regular expressions. Table 6.5 lists some metacharacters and their purpose.

Table 6.5 Metacharacters and their Description and Usage

Metacharacter	Description	Example	Remarks
^	Matches at the beginning of the line.	^Hi	It will match Hi at the start of the string.
$	Matches at the end of the line.	Hi$	It will match Hi at the end of the string.
.	Matches any single character except the newline character.	Lo.	It will match Lot, Log, etc.
[…]	Matches any single character in brackets.	[Hh]ello	It will match "Hello" or "hello".
[^…]	Matches any single character not in brackets.	[^aeiou]	It will match anything other than a lowercase vowel.
re*	Matches 0 or more occurrences of regular expression.	[a-z]*	It will match zero or more occurrence of lowercase characters.
re+	Matches 1 or more occurrence of regular expression.	[a-z]+	It will match one or more occurrence of lowercase characters
re?	Matches 0 or 1 occurrence of regular expression.	Book?	It will match "Book" or "Books".
re{n}	Matches exactly n number of occurrences of regular expression.	42{1}5	It will match 425.
re{n,}	Matches n or more occurrences of regular expression.	42{1,}5	It will match 42225 or any number with more than one 2s between 4 and 5.
re{n,m}	Matches at least n and at most m occurrences of regular expression.	42{1,3}5	It will match 425, 4225, 42225.

(Contd)

Table 6.5 (Contd)

Metacharacter	Description	Example	Remarks
a|b	Matches either a or b.	"Hello" | "Hi"	It will match Hello or Hi.
\w	Matches word characters.	re.search(r'\w', 'xx123xx')	Match will be made.
\W	Matches non-word characters.	if(re.search(r'\W', '@#$%')): print("Done")	Done
\s	Matches whitespace, equivalent to [\t\n\r\f].	if(re.search(r'\s',"abcdsd")): print("Done")	Done
\S	Matches non-whitespace, equivalent to [^\t\n\r\f].	if(re.search(r'\S'," abcdsd")): print("Done")	Done
\d{n}	Matches exactly n digits.	\d{2}	It will match exactly 2 digits.
\d{n,}	Matches n or more digits.	\d{3,}	It will match 3 or more digits.
\d{n,m}	Matches n and at most m digits.	\d{2,4}	It will match 2,3 or 4 digits.
\D	Matches non-digits.	(\D+\d)	It will match Hello 5678, or any string starting with no digit followed by digits(s).
\A	Matches beginning of the string.	\AHi	It will match Hi at the beginning of the string.
\Z	Matches end of the string.	Hi\Z	It will match Hi at the end of the string.

\G	Matches point where last match finished.	import re
		if(re.search(r'\Gabc','abcba cabc')):
		  print("Done")
		else:
		  print("Not Done")
			Not Done

\b	Matches word boundaries when outside brackets. Matches backspace when inside brackets.	\bHi\b	It will match Hi at the word boundary.
\B	Matches non-word boundaries.	\bHi\B	Hi should start at word boundary but end at a non-boundary as in High
\n, \t, etc.	Matches newlines, tabs, etc.	re.search(r'\t', '123 \t abc ')	Match will be made.

6.12.1 Character Classes

When we put the characters to be matched inside square brackets, we call it a character class. For example, `[aeiou]` defines a character class that has a vowel character.

Programming Tip: Placing a ^ at the start of a character class causes it to match any character other than the ones included.

Example 6.31 Program that checks if the string has at least one vowel

```
import re
pattern=r"[aeiou]"
if re.search(pattern,"clue"):
    print("Match clue")
if re.search(pattern,"bcdfg"):
    print("Match bcdfg")
```

OUTPUT

```
Match clue
```

Key points to remember

- Other metacharacters like $ and . have no meaning within character classes. Moreover, the metacharacter ^ has no meaning unless it is the first character in a class.
- Metacharacters like *, +, ?, {, and } specify numbers of repetitions.
- * matches 0 or more occurrences of the regular expression.

Example 6.32 Program to demonstrate the use of metacharacter *

```
import re
pattern=r"hi(de)*"
if re.search(pattern, "hidedededede"):
     print("Match hidedededede")
if re.search(pattern, "hi"):
     print("Match hi")    # zero or more de match
```

OUTPUT

```
Match hidedededede
Match hi
```

- + matches one or more occurrences of the regular expression.

Example 6.33 Program to demonstrate the use of metacharacter +

```
import re
pattern=r"hi(de)+"
if re.search(pattern, "hidedededede"):
      print "Match hidedededede"
if re.search(pattern, "hi"):
      print "Match hi"    # at least one de required for match
```

OUTPUT

```
Match hidedededede
```

- The metacharacter ? means zero or one repetitions.

Example 6.34 Program to demonstrate the use of metacharacter ?

```
import re
pattern=r"hi(de)?"
if re.search(pattern, "hidedededede"):
      print("Match hidedededede")
if re.search(pattern, "hi"):
      print("Match hi")    # matches 0 or 1 occurrence
```

OUTPUT

```
Match hidedededede
Match hi
```

- Curly braces represent the number of repetitions between two numbers. The regular expression {m,n} means m to n repetitions of the expressions. Hence {0,1} is the same as ?. if m is missing, then it is taken to be zero and if n is missing, it is taken to be infinity.

Example 6.35 Program to demonstrate the use of {m,n} regular expression

```
import re
pattern = r"2{1,4}$"
if re.match(pattern,"2"):
    print("Match 2")
if re.match(pattern,"222"):
    print("Match 222")
if re.match(pattern,"22222"):
    print("Match 22222")     # does not match because only max 4 2's will match
```

Programming Tip: metacharacter '+' means {1,}.

OUTPUT

```
Match 2
Match 222
```

Some More Examples

- The pattern ^pr.y$ means that the string should start with pr, then follow with any single character, (except a newline character) and end with y. so the string could be pray or prey.
- The character class [a-z] matches any lowercase character.
- The character class [A-F] matches any uppercase character from A to F.
- The character class [0-9] matches any digit.
- The character class [A-Za-z] defines multiple ranges in one class. It matches a letter of any case.
- The multiple ranges pattern = r"[A-Z][A-Z][0-9]" will match all strings with length 3, where first and second characters are any uppercase character and the third is any digit.
- The metacharacter | means either of the two. For example, pattern r"pr(a|e)y", will match both pray as well as prey.
- The expression match = re.search(r'\d\s*\d\s*\d', 'ab12 3cd') will be matched.
- \s (whitespace) includes newlines characters. To match a run of whitespace that may include a newline character, use \s*.
- [^abc] means any character except "a", "b", or "c" but [a^bc] means an "a", "b", "c", or a "^".

6.12.2 Groups

A group is created by surrounding a part of the regular expression with *parentheses*. You can even give group as an argument to the metacharacters such as * and ?.

Example 6.36 Program to demonstrate the use of groups

```
import re
pattern = r"gr(ea)*t"   # group of ea created
if re.match(pattern,"great"):
    print("Match ea")
if re.match(pattern, "greaeaeaeaeaeaeat"):
    print("Match greaeaeaeaeaeaeat")
```

OUTPUT

```
Match ea
Match greaeaeaeaeaeaeat
```

The content of groups in a match can be accessed by using the `group()` function. For example,
`group(0)` or `group()` returns the whole match.
`group(n)`, where `n` is greater than `0`, returns the n^{th} group from the left.
`group()` returns all groups up from 1.

Example 6.37 Program to demonstrate the use of various group functions

```
import re
pattern = r"Go(od)Go(in)gPy(th)on"
match = re.match(pattern, "GoodGoingPythonGoodGoingPythonGoodGoingPython")
if match:
    print(match.group())
    print(match.group(0))
    print(match.group(1))
    print(match.group(2))
    print(match.groups())
```

OUTPUT

```
GoodGoingPython
GoodGoingPython
od
in
('od', 'in', 'th')
```

Note Python allows you to even nest the groups.

Python supports two useful types of groups—*named group* and *non-capturing group*.

- **Named groups** have the format `(?P<name>...)`, where `name` is the name of the group, and `...` is the content. They are just like normal groups but are accessed by their name as well as by number.

- **Non-capturing groups** having the format `(?:...)` are not accessible by the group method, so they can be added to an existing regular expression without breaking the numbering.

Example 6.38 Program to demonstrate the use of named and non-capturing groups

```
import re
pattern = r"Go(?P<FIRST>od)Go(?:in)gPy(th)on"
match = re.match(pattern, "GoodGoingPythonGoodGoingPythonGoodGoingPython")
if match:
    print(match.group("FIRST"))
    print(match.group(1))
    print(match.group(2))
    print(match.groups())   # (in) is not accessed by group method
```

OUTPUT

```
od
od
th
('od', 'th')
```

Now try the following program and observe the output.

Example 6.39 Program to demonstrate the use of metacharacters and groups

```
import re
match = re.search("([0-9]+).*: (.*)", "Phone number: 12345678, DOB: October 17,
2000")
print(match.group())
print(match.group(1))
print(match.group(2))
print(match.group(1,2))
```

OUTPUT

```
12345678, DOB: October 17, 2000
12345678
October 17, 2000
('12345678', 'October 17, 2000')
```

6.12.3 Application of Regular Expression to Extract Email

We can use regular expressions to extract date, time, email address, etc. from the text. For example, we know that an email address has username which consist of character(s) and may include dots or dashes. The username is followed by @ sign and the domain name. The domain name may also include characters, dashes, and dots. Consider the following email address given below.

```
Info-books@oxford-india.com
```

Now, the regular expression representing the structure of email address can be given as,

```
Pattern = r"[\w.-]+@[\w.-]+"
```

where, [\w.-]+ matches one or more occurrences of (,), dot, or dash.

Example 6.40 Program to extract an email address from a text

```
import re
pattern = r"[\w.-]+@[\w.-]+"
string = "Please send your feedback at info@oxford.com"
match=re.search(pattern, string)
if match:
   print("Email to : ", match.group())
else:
   print("No Match")
```

OUTPUT

```
Email to :  info@oxford.com
```

Note If the string has multiple addresses, use the `re.findall()` method instead of `re.search()` to extract all email addresses.

Program 6.13 Write a program that uses a regular expression to match strings which starts with a sequence of digits (at least one digit) followed by a blank and after this arbitrary characters.

```
import re
pattern = r"^[0-9]+ .*"
string = "12 abc"
match = re.search(pattern, string)
if match:
    print("Match")
```

OUTPUT

```
Match
```

Program 6.14 Write a program to extract each character from a string using a regular expression.

```
import re
result=re.findall(r'.','Good Going')
print(result)
```

Programming Tip: To ignore space use \w instead of \.

OUTPUT

```
['G', 'o', 'o', 'd', ' ', 'G', 'o', 'i', 'n', 'g']
```


Program 6.15 Write a program to extract each word from a string using a regular expression.

```
import re
result=re.findall(r'\w+','Good Going Python')
print(result)
```

OUTPUT

```
['Good', 'Going', 'Python']
```

Program 6.16 Write a program to print the first word of the string.

```
import re
result=re.findall(r'^\w+','Good Going Python')
print(result)
```

OUTPUT

```
['Good']
```

Program 6.17 Write a program to print the last word of the string.

```
import re
result=re.findall(r'\w+$','Good Going Python')
print(result)
```

OUTPUT

```
['Python']
```

Program 6.18 Write a program to print the characters in pairs.

```
import re
result=re.findall(r'\w\w','Good Going Python')
print(result)
```

OUTPUT

```
['Go', 'od', 'Go', 'in', 'Py', 'th', 'on']
```

Program 6.19 Write a program to print only the first two characters of every word.

```
import re
result=re.findall(r'\b\w\w','Good Going Python')
print(result)
```

OUTPUT

```
['Go', 'Go', 'Py']
```

Program 6.20 Write a program to extract a date from a given string.

```
import re
result = re.findall(r'\d{2}-\d{2}-\d{4}','Hello, my name is Srishti and my date of
joining is 11-15-1999 and have experience of more than 17 years')
print("Date of Appointment is : ", result)
```

OUTPUT

```
Date of Appointment is :  ['11-15-1999']
```

Program 6.21 Write a program to extract the year from a given string.

```
import re
result = re.findall(r'\d{2}-\d{2}-(\d{4})','Hello, my name is Srishti and my date
of joining is 11-15-1999 and have experience of more than 17 years')
print("Year of joining is : ", result)
```

OUTPUT

```
Year of joining is :  ['1999']
```

Programming Tip: To print words that begins with consonant use ^.

Program 6.22 Write a program that prints only those words that starts with a vowel.

```
import re
result = re.findall(r'\b[aeiouAEIOU]\w+','Hello, my name is Srishti and my date of
joining is 11-15-1999 and have experience of more than 17 years')
print(result)
```

OUTPUT

```
['is', 'and', 'of', 'is', 'and', 'experience', 'of']
```

Program 6.23 Write a program that validates a mobile phone number. The number should start with 7, 8, or 9 followed by 9 digits.

```
import re
List = ['7838456789', '1234567890', '9876543210', '8901234567', '4567890123']
for i in List:
    result = re.findall(r'[7-9]{1}[0-9]{9}',i)
    if result:
        print(result, end = " ")
```

OUTPUT

```
['7838456789'] ['9876543210'] ['8901234567']
```


Program 6.24 Write a program that replaces ,,;,- from a string with a blank space character.

```
import re
result = re.sub(r'[;,-]',' ','Hello! My name- is Srishti.; My date-of-joining is
11-15-1999 and have experience of, more than 17 years;')
print(result)
```

OUTPUT

```
Hello! My name  is Srishti.  My date of joining is 11 15 1999 and have experience
of  more than 17 years
```

Program 6.25 Write a program that uses a regular expression to pluralize a word.

```
import re
def pluralize(noun):
   if re.search('[sxz]$', noun):
      return re.sub('$', 'es', noun)
   elif re.search('[^aeioudgkprt]h$', noun):
      return re.sub('$', 'es', noun)
   elif re.search('[^aeiou]y$', noun):
      return re.sub('y$', 'ies', noun)
   else:
      return noun + 's'
List = ["bush", "fox", "toy", "cap"]
for i in List:
   print(i, '-', pluralize(i))
```

OUTPUT

```
bush - bushes
fox - foxes
toy - toys
cap - caps
```

Summary

- The Python string data type is a sequence made up of one or more individual characters, where a character could be a letter, digit, whitespace, or any other symbol.
- The `in` operator checks if one character or string is contained in another string.
- The whitespace characters, exclamation mark, and any other symbol (like `?, <,>,*, @, #, $, %`, etc.) that forms a part of the string would be assigned its own index number.
- Concatenate means to join together and append means to add something at the end.
- A raw string literal which is prefixed by an `'r'` passes all the characters as it is.
- Python strings are immutable which means that once created they cannot be changed.
- The number and type of values in the tuple should match the number and type of format sequences or conversion specifications in the string, otherwise an error is returned.
- Strings are an example of Python objects.
- `in` and `not in` operators can be used with strings to determine whether a string is present in another string. Therefore, the `in` and `notin` operator are also known as membership operators.
- The string module consist of a number of useful constants, classes, and functions. These functions are used to manipulate strings.