String Datatype

- string is an ordered collection of characters used to store and represent text-based (Symbols and words (your name), contents of text files, Internet addresses, Python source code etc.) and bytes-based (raw bytes used for media files and network transfer,encoded and decoded forms of non ASCII etc.) information.
- Python strings are immutable sequences.means...
  - the characters they contain have a left-to-right positional order
  - they cannot be changed in place.
- Unicode strings used for dealing with non-ASCII text.
- Unicode is a key tool for those who work in the Internet domain.
- It can pop up in web pages, email content and headers, FTP transfers, GUI APIs, directory tools, and HTML, XML and JSON text.
- strings can be used to represent just about anything that can be encoded as text or bytes.
- strings are easy to use in Python
- there are many ways to write string in our code.for ex.
  - Single quotes: 'spa"m'
  - Double quotes: "spa'm"
  - Triple quotes: '''... spam ...''', """... spam ..."""
  - Escape sequences: "s\tp\na\0m"
  - Raw strings: r"C:\new\test.spm"
  - Bytes strings: b'sp\x01am'
  - Unicode strings: u'eggs\u0020spam'
- Single-Quoted and Double-Quoted Strings Are the Same in python.
- The reason for supporting both is that it allows you to embed a quote character of the other variety inside a string without escaping it with a backslash.
- for Ex.

In [1]:

```
A = 'Hitesh'
B = "Hitesh"
A,B
```

Out[1]:

```
('Hitesh', 'Hitesh')
```

In [2]:

```
A = 'knight"s'
B = "knight's"
A,B
```

Out[2]:

```
('knight"s', "knight's")
```

- `comma` is important between to different strings in python. Without comma Python automatically concatenates adjacent string literals in any expression.
- Adding commas between these strings would result in a `tuple` not a string. Also notice in all of these outputs that Python prints strings in single quotes unless they embed one.

In [3]:

```
string1 = 'hitesh','vaghela'
string1
```

Out[3]:

```
('hitesh', 'vaghela')
```

In [4]:

```
string2 = "Hitesh " 'vaghela'
string2
```

Out[4]:

```
'Hitesh vaghela'
```

Escape Sequences

- scape Sequences Represent Special Characters
- Escape sequences let us embed characters in strings that cannot easily be typed on a keyboard.
- The character \ and one or more characters following it in the string literal, are replaced with a single character in the resulting string object, which has the binary value specified by the escape sequence.
- if Python does not recognize the character after a \ as being a valid escape code it simply keeps the backslash in the resulting string

Escape sequence list

Escape	Meaning
\n	newline
\\	Backslash
\'	Single quote
\"	Double quote
\a	Bell
\b	Backspace
\f	Formfeed
\t	Horizontal tab
\v	Vertical tab
\r	Carriage return

In [5]:

```
s = 'a\nb\tc'
x = "C:\py\code"
print(s)
print(x)
```

```
a
b	c
C:\py\code
```

Limitations and Solution of Escape Sequences

- there are some issues as well of using escape sequences in python.
- for ex.Python newcomer in classes trying to open a file with a filename text.dat that stored in folder named new...
- myfile = open('C:\new\text.dat', 'w')
- they thinking that they will open a file called text.dat in the directory C:\new.
- The problem here is that \n is taken to stand for a newline character, and \t is replaced with a tab.
- In effect the call tries to open a file named C:(newline)ew(tab)ext.dat, with usually less than-stellar results.

- however we can able to handle this situations with vackslash caracters but for new comers to avoid this kind of situations row strings are used.If the letter r (uppercase or lowercase) appears just before the opening quote of a string, it turns off the escape mechanism.
- myfile = open(r'C:\new\text.dat', 'w') # using row string
- myfile = open('C:\\new\\text.dat', 'w') # using backslash

Raw String Limitations

```
- raw string cannot end in a single backslash, because the backslash escapes the following quote character—
you still must escape the surrounding quote character to embed it in the string.
- That is, r"...\" is not a valid string literal—a raw string cannot end in an odd number of backslashes.
- If we need to end a raw string with a single backslash, we can use two and slice off the second
(r'1\nb\tc\\'[:-1]) or tack one on manually (r'1\nb\tc' + '\\')
```

Triple Quoted String

- we also able to create a multiline strings in python using triple quotes.
- triple quoted strings are also useful when we need to embed single and double quotes both in strings.
- triple quoted strings are also called as a `block string`
- Triple-quoted strings are also commonly used for documentation strings as python ignores the strings which are not assign to the variables.python considers this strings as a comments.
- thus triple-quoted strings are sometimes used as a “horribly hackish” way to temporarily disable lines of code during development

In [6]:

```
St = """Always look
 on the bright
 side of life."""
St
```

Out[6]:

```
'Always look\n on the bright\n side of life.'
```

In [7]:

```
X = 1
"""
import os # Disable this code temporarily
print(os.getcwd())
"""
Y = 2
print(X)
```

```
1
```

lenght of string

- built-in len function returns the actual number of characters in a string.

In [8]:

```
String1 = "Hitesh"
String2 = "Hitesh\tVaghela"
print(String1)
print(String2)
```

```
Hitesh
Hitesh  Vaghela
```

In [9]:

```
len(String1)
```

Out[9]:

6

In [10]:

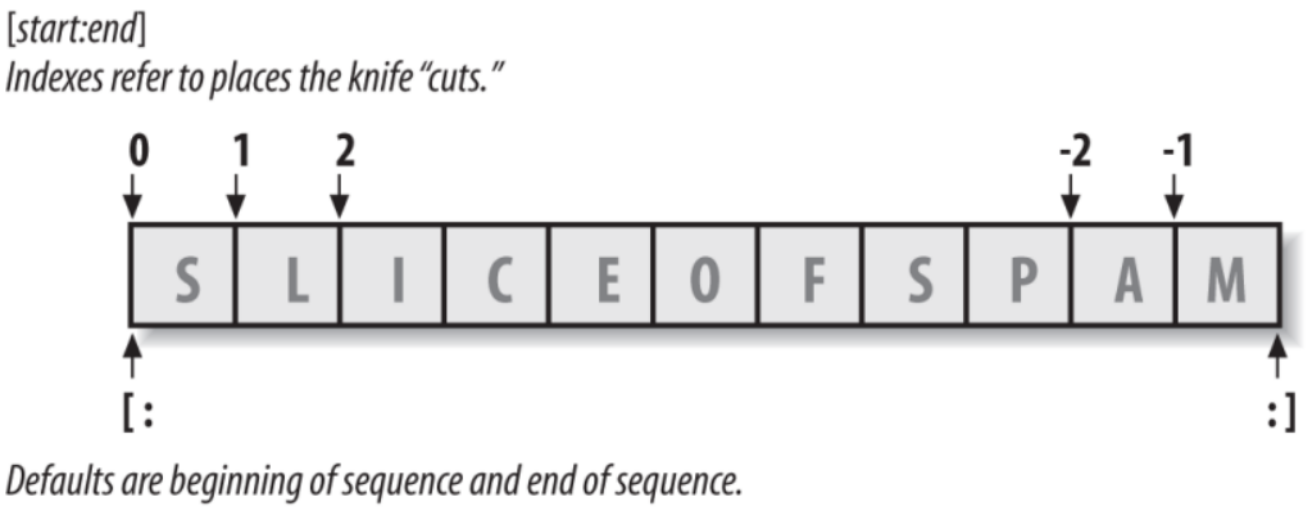
```
len(String2)
```

Out[10]:

14

Indexing and Slicing

- Slicing is Used to access the elements of string. we use index for slicing.Index is nothing but a position of elements.
- In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on
- In Python characters in a string are fetched by indexing providing the numeric offset of the desired component in square brackets after then string.
- In Python we can also index backward from the end
  - positive indexes count from the left
  - negative indexes count back from the right



In [11]:

```
S = "Hitesh Vaghela"
S[0] # The first item in S, indexing by zero-based position
```

Out[11]:

'H'

In [12]:

```
S[1] # The second item from the left
```

Out[12]:

'i'

In [13]:

```
S[-1] # The last item from the end in S
```

Out[13]:

```
'a'
```

In [14]:

```
S[-2] # The second-to-last item from the end
```

Out[14]:

```
'l'
```

- technically a negative index is simply added to the string's length so the following two operations are equivalent

In [15]:

```
S[-1] # The last item in S
```

Out[15]:

```
'a'
```

In [16]:

```
S[len(S)-1] # Negative indexing, the hard way
```

Out[16]:

```
'a'
```

- sequences also support a more general form of indexing known as slicing.
- which is a way to extract an entire section (slice) in a single step. For example...
- general form of slicing is X[I:J]
- it returns everything in X from offset I up to offset J but not including offset J.
- The result is returned in a new object.
- In a slicing....
  - the left bound defaults to zero
  - the right bound defaults to the length of the sequence being sliced.
- Here... also we have to note that as string are immutable it creates a slice as a new object until we assign to the old object.
- Extended slicing (S[i:j:k]) accepts a step (or stride) k, which defaults to +1.

In [17]:

```
S = "Hitesh vaghela Loves Reshmma Vaghela"
S[0:15] # it Slice of S from offsets 1 to offsets 15. here...first offset is included and last offset is exc
```

Out[17]:

```
'Hitesh vaghela '
```

In [18]:

```
S[1:] # Everything past the first (1:len(S))
```

Out[18]:

```
'itesh vaghela Loves Reshmma Vaghela'
```

In [19]:

```
S[-16:] #fetches items at offset 16 from right to end of string.
```

Out[19]:

```
' Reshmma Vaghela'
```

In [20]:

```
S[:14] # Everything past the first (0:15)
```

Out[20]:

```
'Hitesh vaghela'
```

In [21]:

```
S[:-2] # fetches items at offset 0 up to but not including the last 2 items.
```

Out[21]:

```
'Hitesh vaghela Loves Reshmma Vaghe'
```

In [22]:

```
S[:] # Everything past the first (0:len(S))
# this is also used to making a top-level copy of Sequence.
```

Out[22]:

```
'Hitesh vaghela Loves Reshmma Vaghela'
```

In [23]:

```
X = "Purvi Vaghela"
X[::2] # gets every second item from the beginning to the end of the sequence.
```

Out[23]:

```
'PriVgea'
```

In [24]:

```
X = "Purvi Vaghela"
X[::-1] # gets every item from the end to the beginning of the sequence with 1 skip.
# this is also  used for revesing the string
```

Out[24]:

```
'alehgaV ivruP'
```

In [25]:

```
X[::-2] # gets every second item from the beginning to the end of the sequence.
```

Out[25]:

```
'aegVirP'
```

In [26]:

```
S = 'aabcdefghijklmnopqrstuvwxyz'
S[1:20:2]
```

Out[26]:

```
'acegikmoqs'
```

In [27]:

```
S[::3]
```

Out[27]:

```
'acfilorux'
```

String Concatination

- as sequences strings also support concatenation with the plus sign and repetition with star sign.
- concatenation means joining two strings into a new string.
- repetition means making a new string by repeating string many times.
- as we know that string is immutable so it creates as new object as a result. and old object remain as it is until we assign another value to it.
- there are mainly three ways to concatinate strings...
  - the plus `(+)` operator concates two strings.
  - The `__add__` method of strings performs concatenation.
  - we can also add a string using add a space `" "` between them.

In [28]:

```
S = "VAGHELA"
S + ' HITESH'
```

Out[28]:

```
'VAGHELA HITESH'
```

In [29]:

```
S # original string S is unchanged as it creates a new object each time.
```

Out[29]:

```
'VAGHELA'
```

In [30]:

```
S = S + ' HITESH '
S
```

Out[30]:

```
'VAGHELA HITESH '
```

In [31]:

```
A = 'Hitesh'
A.__add__(' Vaghela')
```

Out[31]:

```
'Hitesh Vaghela'
```

In [32]:

```
A = 'Purvi'
B = 'Reshma'
A +" " + B
```

Out[32]:

```
'Purvi Reshma'
```

- String and Number both are different datatype so we can't mix strings and number types around operators such as + we can manually convert operands before that operation if needed.
- in Python...

- the meaning of an operation depends on the objects being operated on.
- the plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings.
- This property of Python is called as a polymorphism.
- thus 'abc'+9 raises an error instead of automatically converting 9 to a string.

In [33]:

```
S = "42"
I = 1
S + I
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 3
      1 S = "42"
      2 I = 1
----> 3 S + I

TypeError: can only concatenate str (not "int") to str
```

In [34]:

```
int(S) + I # Force addition
```

Out[34]:

```
43
```

In [35]:

```
S + str(I) # Force concatenation
```

Out[35]:

```
'421'
```

String Repitation

- star (*) operator is used to perform repitation in string.

In [36]:

```
S = "Hardik "
S * 8
```

Out[36]:

```
'Hardik Hardik Hardik Hardik Hardik Hardik Hardik Hardik '
```

String modification

- strings are immutable datatype.
- we can never overwrite the values of immutable objects.
- beacause of this we were not changing the original string with any of the operations we ran on it.
- Every string operation is defined to produce a new string as its result.

In [37]:

```
S = 'Spam'
S[0] = 'z' # Immutable objects cannot be changed
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[37], line 2
      1 S = 'Spam'
----> 2 S[0] = 'z' # Immutable objects cannot be changed

TypeError: 'str' object does not support item assignment
```

In [38]:

```
 # But we can run expressions to make new objects
S = 'z' + S[1:]
S
```

Out[38]:

```
'zpam'
```

replace() method

- we can achieve similar effects with string method named replace().
- The replace() method replaces a specified phrase with another specified phrase.
- All occurrences of the specified phrase will be replaced, if count is not specified.
- Parameter...
  - oldvalue Required. The string to search for
  - newvalue Required. The string to replace the old value with
  - count Optional. A number specifying how many occurrences of the old value you want to replace. Default is all occurrences

In [39]:

```
A = "Hixesh"
A.replace('x','t') # creates a new object
```

Out[39]:

```
'Hitesh'
```

In [40]:

```
A
```

Out[40]:

```
'Hixesh'
```

In [41]:

```
A = A.replace('x','t')
A
```

Out[41]:

```
'Hitesh'
```

In [42]:

```
txt = "one one was a race horse, two two was one too."
txt.replace("one", "three")
```

Out[42]:

```
'three three was a race horse, two two was three too.'
```

In [43]:

```
txt = "one one was a race horse, two two was one too."
txt.replace("one", "three",2)
```

Out[43]:

```
'three three was a race horse, two two was one too.'
```

capitalize() method

- The capitalize() method returns a string where the first character is upper case and the rest is lower case.
- if there is a any upper case character other than fitst charecter then they are converted to lower case.

In [44]:

```
txt1 = "hitesh vaghela"
txt1.capitalize()
```

Out[44]:

```
'Hitesh vaghela'
```

In [45]:

```
txt2 = "python is FUN!"
txt2.capitalize()
```

Out[45]:

```
'Python is fun!'
```

In [46]:

```
txt3 = "36 Is My AGE."
txt3.capitalize()
```

Out[46]:

```
'36 is my age.'
```

casefold() method

- The casefold() method returns a string where all the characters are lower case.
- This method is similar to the lower() method, but the casefold() method is stronger, more aggressive.

In [47]:

```
txt = "Hello My NAME IS HITESH VAGHELA"
txt.casefold()
```

Out[47]:

```
'hello my name is hitesh vaghela'
```

lower() method, upper() method, title() method

- lower() method Converts a string into lower case.
- upper() method Converts a string into upper case.
- title() method Converts the first character of each word to upper case.
- The swapcase() method returns a string where all the upper case letters are lower case and vice versa.

In [48]:

```
S = 'spam'
S.upper()
```

Out[48]:

```
'SPAM'
```

In [49]:

```
S = 'SPAM'
S.lower()
```

Out[49]:

```
'spam'
```

In [50]:

```
S = 'hitesh vaghela'
S.title()
```

Out[50]:

```
'Hitesh Vaghela'
```

In [51]:

```
S = "Hitesh Vaghela"
S.swapcase()
```

Out[51]:

```
'hITESH vAGHELA'
```

index() and rindex() method

- The index() method finds the first occurrence of the specified value.
- The index() method raises an exception if the value is not found.
- The index() method is almost the same as the find() method, the only difference is that the find() method returns -1 if the value is not found.
- Parameter...
  - value Required. The value to search for
  - start Optional. Where to start the search. Default is 0
  - end Optional. Where to end the search. Default is to the end of the string

In [52]:

```
A = "Hello and welcome to my Youtube Channel."
A.index("e")
```

Out[52]:

```
1
```

In [53]:

```
A.index("d", 5, 10)
```

Out[53]:

```
8
```

In [54]:

```
A.index("X")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[54], line 1
----> 1 A.index("X")

ValueError: substring not found
```

In [55]:

```
A.index("e", 5, 10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[55], line 1
----> 1 A.index("e", 5, 10)

ValueError: substring not found
```

- The rindex() method finds the last occurrence of the specified value.
- The rindex() method raises an exception if the value is not found.
- The rindex() method is almost the same as the rfind() method.
- Parameter...
  - value Required. The value to search for
  - start Optional. Where to start the search. Default is -1.
  - end Optional. Where to end the search. Default is to the stsrt of the string

In [56]:

```
A = "Hello and welcome to my Youtube Channel."
A.rindex("e")
```

Out[56]:

```
37
```

In [57]:

```
A.rindex("d", 5, 10)
```

Out[57]:

```
8
```

In [58]:

```
A.rindex("X")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[58], line 1
----> 1 A.rindex("X")

ValueError: substring not found
```

find() and rfind() method

- The find() method finds the first occurrence of the specified value.
- The find() method returns -1 if the value is not found.
- The find() method is almost the same as the index() method, the only difference is that the index() method raises an exception if the value is not found.

- Parameter...
  - value Required. The value to search for
  - start Optional. Where to start the search. Default is 0
  - end Optional. Where to end the search. Default is to the end of the string

In [59]:

```
A = "Hello and welcome to my Youtube Channel."
A.find("b")
```

Out[59]:

```
29
```

In [60]:

```
A.find("X")
```

Out[60]:

```
-1
```

- The rfind() method finds the last occurrence of the specified value.
- The rfind() method returns -1 if the value is not found.
- The rfind() method is almost the same as the rindex() method.
- Parameter...
  - value Required. The value to search for
  - start Optional. Where to start the search. Default is -1
  - end Optional. Where to end the search. Default is to the start of the string

In [61]:

```
A = "Hello and welcome to my Youtube Channel."
A.rfind("e")
```

Out[61]:

```
37
```

In [62]:

```
A.rfind("X")
```

Out[62]:

```
-1
```

ord() and chr() Method

- string method ord used to convert a character and number into ASCII values.
- The chr method performs the inverse operation of ord method taking an integer code and converting it to the corresponding character.
- These tools can also be used to perform a sort of string-based math.

In [63]:

```
ord('\n') # \n is a byte with the binary value 10 in ASCII
```

Out[63]:

```
10
```

In [64]:

```
ord('H')
```

Out[64]:

```
72
```

In [65]:

```
chr(10)
```

Out[65]:

```
'\n'
```

In [66]:

```
chr(72)
```

Out[66]:

```
'H'
```

In [67]:

```
S = '5'
S = chr(ord(S) + 1)
S
```

Out[67]:

```
'6'
```

In [68]:

```
S = "H"
S = chr(ord(S) + 1)
S
```

Out[68]:

```
'I'
```

count() method

- The count() method returns the number of times a specified value appears in the string.
- parameters...
- value Required. A String. The string to value to search for
- start Optional. An Integer. The position to start the search. Default is 0
- end Optional. An Integer. The position to end the search. Default is the end of the string

In [69]:

```
A = "I love Reshma. Reshma is a very beautiful girl."
A.count("Reshma")
```

Out[69]:

```
2
```

In [70]:

```
A.count("Reshma",3,14)
```

Out[70]:

```
1
```

In [71]:

```
A.count("e",3,24)
```

Out[71]:

```
3
```

ljust(), center() and rjust() Methods

- The ljust() method will left align the string, using a specified character (space is default) as the fill character.
- The center() method will center align the string, using a specified character (space is default) as the fill character.
- The rjust() method will right align the string, using a specified character (space is default) as the fill character.
- Parameter...
  - length Required. The length of the returned string
  - character Optional. A character to fill the missing space (to the left of the string). Default is " " (space).

In [72]:

```
txt = "banana"
txt.ljust(20)
```

Out[72]:

```
'banana              '
```

In [73]:

```
txt = "banana"
txt.ljust(20,'-')
```

Out[73]:

```
'banana--------------'
```

In [74]:

```
A = "banana"
A.center(20)
```

Out[74]:

```
'       banana       '
```

In [75]:

```
A = "banana"
A.center(20, "-")
```

Out[75]:

```
'-------banana-------'
```

In [76]:

```
txt = "banana"
txt.rjust(20)
```

Out[76]:

```
'              banana'
```

In [77]:

```
txt = "banana"
txt.rjust(20,'-')
```

Out[77]:

```
'--------------banana'
```

zfill() Method

- The zfill() method adds zeros at the beginning of the string, until it reaches the specified length.
- if lenght is grater than specified lenght then simply it returns string as it is.

In [78]:

```
a = "hello"
a.zfill(10)
```

Out[78]:

```
'00000hello'
```

In [79]:

```
c = "10.000"
c.zfill(10)
```

Out[79]:

```
'000010.000'
```

In [80]:

```
b = "welcome to the jungle"
b.zfill(10)
```

Out[80]:

```
'welcome to the jungle'
```

split(), splitlines() and rsplit() method

- The split() method splits a string into a list.
- we can specify the separator, default separator is any whitespace.
- Parameters...
  - separator Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator
  - maxsplit Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences"
- When maxsplit is specified, the list will contain the specified number of elements plus one.
- The splitlines() method splits a string into a list. The splitting is done at line breaks.
- The rsplit() method also splits a string into a list,but starting from the right.
- If no "max" is specified, this method will return the same as the split() method.
- When maxsplit is specified, the list will contain the specified number of elements plus one.

In [81]:

```
txt = "welcome to the jungle"
txt.split()
```

Out[81]:

```
['welcome', 'to', 'the', 'jungle']
```

In [82]:

```
txt = "hitesh,reshma,purvi,devanshi,sweta,pooja,anjana,shilpa"
txt.split(",")
```

Out[82]:

```
['hitesh', 'reshma', 'purvi', 'devanshi', 'sweta', 'pooja', 'anjana', 'shilpa']
```

In [83]:

```
txt = "hitesh,reshma,purvi,devanshi,sweta,pooja,anjana,shilpa"
txt.split(",",6)
```

Out[83]:

```
['hitesh', 'reshma', 'purvi', 'devanshi', 'sweta', 'pooja', 'anjana,shilpa']
```

In [84]:

```
txt = "welcome to the jungle"
txt.rsplit()
```

Out[84]:

```
['welcome', 'to', 'the', 'jungle']
```

In [85]:

```
txt = "hitesh,reshma,purvi,devanshi,sweta,pooja,anjana,shilpa"
txt.rsplit(",")
```

Out[85]:

```
['hitesh', 'reshma', 'purvi', 'devanshi', 'sweta', 'pooja', 'anjana', 'shilpa']
```

In [86]:

```
txt = "hitesh,reshma,purvi,devanshi,sweta,pooja,anjana,shilpa"
txt.rsplit(",",6)
```

Out[86]:

```
['hitesh,reshma', 'purvi', 'devanshi', 'sweta', 'pooja', 'anjana', 'shilpa']
```

In [87]:

```
txt = "Thank you for the music\nWelcome to the jungle"
txt.splitlines()
```

Out[87]:

```
['Thank you for the music', 'Welcome to the jungle']
```

In [88]:

```
txt = "Thank you for the music\nWelcome to the jungle"
txt.splitlines(True)
```

Out[88]:

```
['Thank you for the music\n', 'Welcome to the jungle']
```

partition() and rpartition() method

- The partition() method searches for a specified string, and splits the string into a tuple containing three elements.
- The first element contains the part before the specified string.
- The second element contains the specified string.
- The third element contains the part after the string.
- This method searches for the first occurrence of the specified string.

- If the specified value is not found the partition() method returns a tuple containing:
    1. the whole string
    2. an empty string
    3. an empty string

In [89]:

```
txt = "I could eat bananas all day.bananas are my favorite fruit"
txt.partition("bananas")
```

Out[89]:

```
('I could eat ', 'bananas', ' all day.bananas are my favorite fruit')
```

In [90]:

```
txt = "I could eat bananas all day"
txt.partition("apples")
```

Out[90]:

```
('I could eat bananas all day', '', '')
```

- The rpartition() method searches for the last occurrence of a specified string, and splits the string into a tuple containing three elements.
- The first element contains the part before the specified string.
- The second element contains the specified string.
- The third element contains the part after the string.
- If the specified value is not found the rpartition() method returns a tuple containing...
    1. an empty string
    2. an empty string
    3. the whole string

In [91]:

```
txt = "I could eat bananas all day.bananas are my favorite fruit"
txt.rpartition("bananas")
```

Out[91]:

```
('I could eat bananas all day.', 'bananas', ' are my favorite fruit')
```

In [92]:

```
txt = "I could eat bananas all day"
txt.rpartition("apples")
```

Out[92]:

```
('', '', 'I could eat bananas all day')
```

join() method

- The join() method takes all items in an iterable and joins them into one string.
- A string must be specified as the separator.

In [93]:

```
myTuple = ("John", "Peter", "Vicky")
" ".join(myTuple)
```

Out[93]:

```
'John Peter Vicky'
```

In [94]:

```
mylist = ["John", "Peter", "Vicky"]
"".join(myTuple)
```

Out[94]:

```
'JohnPeterVicky'
```

In [95]:

```
A = ['A','B','C']
Sep = " "
new_str = Sep.join(A)
new_str
```

Out[95]:

```
'A B C'
```

In [96]:

```
D = {"name": "John", "country": "Norway"}
Sep = "#"
x = Sep.join(D)
x
```

Out[96]:

```
'name#country'
```

encode() and decode() method

- The encode() method encodes the string, using the specified encoding.
- The decode() method decodes the string, using the specified encoding.
- default encoding is UTF-8.other encodings are...
  - ascii
  - UTF-16
- default errrors is Strict.other errors...
  - 'backslashreplace' - uses a backslash instead of the character that could not be encoded
  - 'ignore' - ignores the characters that cannot be encoded
  - 'namereplace' - replaces the character with a text explaining the character
  - 'strict' - Default, raises an error on failure
  - 'replace' - replaces the character with a questionmark
  - 'xmlcharrefreplace' - replaces the character with an xml character

In [97]:

```
txt = "My name is Hitesh"
print(txt.encode(encoding="UTF-8",errors="backslashreplace"))
print(txt.encode(encoding="UTF-16",errors="ignore"))
```

```
b'My name is Hitesh'
b'\xff\xfeM\x00y\x00 \x00n\x00a\x00m\x00e\x00 \x00i\x00s\x00 \x00H\x00i\x00t\x00e\x00s\x00h\x0
0'
```

In [98]:

```
B = "Hitesh"
B = B.encode("UTF-16","replace")
B
```

Out[98]:

```
b'\xff\xfeH\x00i\x00t\x00e\x00s\x00h\x00'
```

In [99]:

```
B.decode("UTF-16","replace") # Translate to normal string
```

Out[99]:

```
'Hitesh'
```

strip(), lstrip(), rstrip() method

- The strip() method removes any leading, and trailing whitespaces.
- Leading means at the beginning of the string, trailing means at the end.
- we can specify which characters to remove, if not, any whitespaces will be removed.
- The lstrip() method removes any leading characters (space is the default leading character to remove).
- The rstrip() method removes any trailing characters (space is the default trailing character to remove).

In [100]:

```
txt = "     banana     "
txt.strip()
```

Out[100]:

```
'banana'
```

In [101]:

```
txt = ",,,,,rrttgg.....banana....rrr"
txt.strip(",.grt")
```

Out[101]:

```
'banana'
```

In [102]:

```
txt = "     banana     "
txt.lstrip()
```

Out[102]:

```
'banana     '
```

In [103]:

```
txt = ",,,,,rrttgg.....banana....rrr"
txt.lstrip(",.grt")
```

Out[103]:

```
'banana....rrr'
```

In [104]:

```
txt = "     banana     "
txt.rstrip()
```

Out[104]:

```
'     banana'
```

In [105]:

```
txt = ",,,,,rrttgg.....banana....rrr"
txt.rstrip(",.grt")
```

Out[105]:

```
',,,,,rrttgg.....banana'
```

startswith() and endswith() method

- The startswith() method returns True if the string starts with the specified value, otherwise False.
- The endswith() method returns True if the string ends with the specified value, otherwise False.

In [106]:

```
txt = "Hello, welcome to my world."
txt.startswith("Hello")
```

Out[106]:

```
True
```

In [107]:

```
txt = "Hello, welcome to my world."
txt.startswith("w",7,21)
```

Out[107]:

```
True
```

In [108]:

```
txt = "Hello, welcome to my world."
txt.endswith("my world.")
```

Out[108]:

```
True
```

In [109]:

```
txt = "Hello, welcome to my world."
txt.endswith("my world.", 5, 11)
```

Out[109]:

```
False
```

Other String Methods

- The isalnum() method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9).
- The isalpha() method returns True if all the characters are alphabet letters (a-z)
- The isascii() method returns True if all the characters are ascii characters (a-z).
- The isdecimal() method returns True if all the characters are decimals (0-9).
- The isdigit() method returns True if all the characters are digits, otherwise False.
- The isnumeric() method returns True if all the characters are numeric (0-9), otherwise False.
  - Exponents, like ² and ¾ are also considered to be numeric values.
  - "-1" and "1.5" are NOT considered numeric values, because all the characters in the string must be numeric, and the - and the . are not.
- The isspace() method returns True if all the characters in a string are whitespaces, otherwise False.
- The isupper() method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.
- The islower() method returns True if all the characters are in lower case, otherwise False.Numbers, symbols and spaces are not checked, only alphabet characters.
- The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False. Symbols and numbers are ignored.

In [110]:

```
txt = "Company12"
txt.isalnum()
```

Out[110]:

```
True
```

In [111]:

```
txt = "Company10"
txt.isalpha()
```

Out[111]:

```
False
```

In [112]:

```
txt = "Company123"
txt.isascii()
```

Out[112]:

```
True
```

In [113]:

```
txt = "1234"
txt.isdecimal()
```

Out[113]:

```
True
```

In [114]:

```
a = "\u0030" #unicode for 0
b = "\u0047" #unicode for G
print(a.isdecimal())
print(b.isdecimal())
```

```
True
False
```

In [115]:

```
txt = "50800"
txt.isdigit()
```

Out[115]:

```
True
```

In [116]:

```
a = "\u0030" #unicode for 0
b = "\u00B2" #unicode for ²

print(a.isdigit())
print(b.isdigit())
```

```
True
True
```

In [117]:

```
a = "\u0030" #unicode for 0
b = "\u00B2" #unicode for &sup2;
c = "10km2"
d = "-1"
e = "1.5"

print(a.isnumeric())
print(b.isnumeric())
print(c.isnumeric())
print(d.isnumeric())
print(e.isnumeric())
```

```
True
True
False
False
False
```

In [118]:

```
txt = "   "
txt.isspace()
```

Out[118]:

```
True
```

In [119]:

```
txt = "   s   "
txt.isspace()
```

Out[119]:

```
False
```

In [120]:

```
a = "Hello World!"
b = "hello 123"
c = "MY NAME IS PETER"

print(a.isupper())
print(b.isupper())
print(c.isupper())
```

```
False
False
True
```

In [121]:

```
a = "Hello world!"
b = "hello 123"
c = "mynameisPeter"

print(a.islower())
print(b.islower())
print(c.islower())
```

```
False
True
False
```

In [122]:

```
a = "HELLO, AND WELCOME TO MY WORLD"
b = "Hello"
c = "22 Names"
d = "This Is %'!?"

print(a.istitle())
print(b.istitle())
print(c.istitle())
print(d.istitle())
```

```
False
True
True
True
```

String formating

- There are five different ways to perform string formatting in Python
  - Formatting with % Operator.
  - Formatting with format() string method.
  - Formatting with string literals, called f-strings.
  - Formatting with String Template Class
  - Formatting with center() string method.

Formatting string using % Operator

- oldest method of string formatting.
- we use the modulo % operator.
- The modulo % is also known as the “string-formatting operator”.
- we also able to insert Multiple Strings using the modulo Operator

In [123]:

```
print("Hitesh is a %s boy"%"good")
```

```
Hitesh is a good boy
```

In [124]:

```
print("Hitesh is a %s boy.he loves %s < %d." % ("good",'Reshma',3))
```

```
Hitesh is a good boy.he loves Reshma < 3.
```

In [125]:

```
X = 'Good'
Y = 'Reshma'
Z = 3
W = 'Hitesh'
print("Hitesh is a %s boy.he loves %s < %d.she also likes %s < %5.2f" % (X,Y,Z,W,Z))
```

```
Hitesh is a Good boy.he loves Reshma < 3.she also likes Hitesh <  3.00
```

- `%` operator also usesd for Precision Handling in Python.
- Floating-point numbers use the format `%A.Bf` .
  - `A` would be the minimum number of digits to be present in the string.if the whole number doesn’t have this many digits it will be padded with white space.
  - `Bf` represents how many digits are to be displayed after the decimal point.

In [126]:

```
'%5.4f' %(3.141592),len('%5.4f' %(3.141592))
```

Out[126]:

```
('3.1416', 6)
```

In [127]:

```
'%8.2f' %(3.141592),len('%8.2f' %(3.141592))
```

Out[127]:

```
('    3.14', 8)
```

Formatting using format() Method

- Format() method was introduced with Python3 for handling complex string formatting more efficiently.
- Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces { } into a string and calling the str.format().
- The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.
- Syntax: 'String here {} then also {}'.format('something1','something2')
- it also supports an Index-based Insertion.
- curly braces {} with indices are used within the string '{2} {1} {0}' to indicate the positions where the corresponding values will be placed.we also able to give custom indices.
- curly braces {} with named placeholders ({a}, {b}, {c}) are used within the string 'a: {a}, b: {b}, c: {c}' to indicate the positions where the corresponding named arguments will be placed.

In [128]:

```
print('We all are {}.'.format('equal'))
```

```
We all are equal.
```

In [129]:

```
print('{2} {1} {0}'.format('directions','the', 'Read'))
```

```
Read the directions
```

In [130]:

```
print('a: {a}, b: {b}, c: {c}'.format(a = 1,b = 'Two',c = 12.3))
```

```
a: 1, b: Two, c: 12.3
```

In [131]:

```
print('a: {a}, b: {a}, c: {a}'.format(a = 1))
```

```
a: 1, b: 1, c: 1
```

In [132]:

```
a = 1
b = 'Two'
c = 12.3
print('a: {a}, b: {b}, c: {c}'.format(a,b,c))
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[132], line 4
      2 b = 'Two'
      3 c = 12.3
----> 4 print('a: {a}, b: {b}, c: {c}'.format(a,b,c))

KeyError: 'a'
```

- this method also supports a Float Precision.
- Syntax: {[index]:[width][.precision][type]}
- The type can be used with format codes:
  - ‘d’ for integers
  - ‘f’ for floating-point numbers
  - ‘b’ for binary numbers
  - ‘o’ for octal numbers
  - ‘x’ for octal hexadecimal numbers
  - ‘s’ for string
  - ‘e’ for floating-point in an exponent format

In [133]:

```
'{0:1.5f}'.format(3.141592), len('{0:1.5f}'.format(3.141592))
```

Out[133]:

```
('3.14159', 7)
```

In [134]:

```
'{0:8.2f}'.format(3.141592), len('{0:8.2f}'.format(3.141592))
```

Out[134]:

```
('    3.14', 8)
```

In [135]:

```
'{:.2f}'.format(3.141592), len('{:.2f}'.format(3.141592)) # index always 0 and default width lenght of numbe
```

Out[135]:

```
('3.14', 4)
```

String Formatting with F-Strings

- PEP-498 introduced a new string formatting mechanism known as `Literal String Interpolation` or `F-strings` (because of the leading f character preceding the string literal).
- The idea behind f-String in Python is to make string interpolation simpler.
- To create an f-string in Python we need to add the prefix "f" to the string.
- F-strings provide a concise and convenient way to embed Python expressions inside string literals for formatting.
- we can also able to insert Python expressions in string and perform arithmetic operations in it.
- We can also use lambda expressions in f-string formatting.
- Float precision in the f-String Method
- Syntax: {value:{width}.{precision}}

In [136]:

```
name = 'Hitesh'
print(f"My name is {name}.")
```

```
My name is Hitesh.
```

In [137]:

```
name = 'Hitesh'
name2 = 'Reshma'
num = 3
print(f"My name is {name}.i love {name2} < {num}")
```

```
My name is Hitesh.i love Reshma < 3
```

In [138]:

```
a = 5
b = 10
print(f"He said his age is {2 * (a + b)}.")
```

```
He said his age is 30.
```

In [139]:

```
a = 10
b = 4
print(f"He said his age is { (lambda x: x*a) (b) }")
```

```
He said his age is 40
```

In [140]:

```
num = 3.14159
f"{num:{1}.{5}}" , len(f"{num:{1}.{5}}")
```

Out[140]:

```
('3.1416', 6)
```

In [141]:

```
num = 3.14159
f"{num:{9}.{3}}" , len(f"{num:{9}.{3}}")
```

Out[141]:

```
('     3.14', 9)
```

String Methods for Membership and other Testing

In [142]:

```
S = 'Hitesh'
'X' in S,"H" in S
```

Out[142]:

```
(False, True)
```

In [143]:

```
S1 = 'Hitesh'
S2 = 'Hitesh'
S1 == S2,S1 is S2 # same object because of cache
```

Out[143]:

```
(True, True)
```

In [144]:

```
S1 = 'Hitesh Vaghela'
S2 = 'Hitesh Vaghela'
S1 == S2,S1 is S2 # different object as cache size is 2Mb and only small object will be catched
```

Out[144]:

```
(True, False)
```

Usefull Tricks

- leading and trailing double underscores is the naming pattern Python uses for implementation details.
- The names without the underscores in this list are the callable methods on string objects.
- The dir function simply gives the methods' names.
- To ask what that methods do we can pass them to the help function
- `shift + tab` opens a documentation of method in jupter notebook

In [145]:

```
S = "A"
print(dir(S))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__for
mat__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul_
_', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '_
_setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnu
m', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isp
rintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartitio
n', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'tr
anslate', 'upper', 'zfill']
```

In [146]:

```
help(S.upper)
```

```
Help on built-in function upper:

upper() method of builtins.str instance
    Return a copy of the string converted to uppercase.
```