Strings

A string is simply a series of characters. Strings are used to record the text information such as name. This is one of the important features of the Python language.

Creating a String

Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings.

In [ ]:

```
# String build with a single quote
'This is a string'
```

Out[ ]:

```
'This is a string'
```

In [ ]:

```
# String build with a double quote
"This is also a string"
```

Out[ ]:

```
'This is also a string'
```

This flexibility allows you to use quotes and apostrophes within your strings:

In [ ]:

```
'I told my friend, "Python is my favorite language!"'
```

Out[ ]:

```
'I told my friend, "Python is my favorite language!"'
```

In [ ]:

```
"One of Python's strengths is its diverse and supportive community."
```

Out[ ]:

```
"One of Python's strengths is its diverse and supportive community."
```

Before going deeper into the Strings first let us know the Variables

Variables

So, what is a variable? Consider that your house needs a name. You place a nameplate at the front gate of your house. People will now recognize your house through that nameplate. That nameplate can be considered as variable. Like a nameplate points to the house, a variable points to the value that is stored in memory. When you create a variable, the interpreter will reserve some space in the memory to store values.

In [ ]:

```
# Assign s as a string
s = 'Hello Python'
```

In [ ]:

```
#Check
s
```

Out[ ]:

```
'Hello Python'
```

In [ ]:

```
# Print the object
print(s)
```

```
Hello Python
```

In the preceding example, a "Hello Python" string has been assigned to an s variable. By using the ID function, we obtained the memory address.

In [ ]:

```
id(s)
```

Out[ ]:

```
1879279315120
```

In order to find the length of the string, the len() function is used.

In [ ]:

```
len(s)
```

Out[ ]:

```
12
```

String Indexing

We know strings are a sequence, which means Python can use indexes to call all the sequence parts. Let's learn how String Indexing works. We use brackets [] after an object to call its index.

In [ ]:

```
name = "Shivam Singh"
```

In [ ]:

```
name[0]
```

Out[ ]:

```
'S'
```

In [ ]:

```
len(name)
```

Out[ ]:

```
12
```

In [ ]:

```
name[11]
```

Out[ ]:

```
'h'
```

In [ ]:

```
name[12]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-17-25aa37574f4e> in <module>
----> 1 name[12]

IndexError: string index out of range
```

The "Shivam Singh" string is 12 characters long, which means it ranges from 0 to 11 index. The name[0] represents the character 'S'. If you give the 12th index value, then the Python interpreter generates an error out of range.

Let's see what is reverse indexing:

In [ ]:

```
name[-1]
```

Out[ ]:

```
'h'
```

In [ ]:

```
name[-12]
```

Out[ ]:

```
'S'
```

The -1 index represents the last character and -12 represents the first character. In the computer world, the computer counts the index from 0 itself. The following diagram will clear all your doubts:

Slicing

In many situations, you might need a particular portion of strings such as the first three characters of the string. We can use a : to perform slicing which grabs everything up to a designated point.

In [ ]:

```
# Grab everything past the first term all the way to the length of name which is len(name
)
name[1:]
```

Out[ ]:

```
'hivam Singh'
```

In [ ]:

```
# Note that there is no change to the original name
name
```

Out[ ]:

```
'Shivam Singh'
```

In [ ]:

```
# Grab everything UP TO the 3rd index
name[:3]
```

Out[ ]:

```
'Shi'
```

In [ ]:

```
# Grab eveything from 4th index to 9th index
name[4:10]
```

Out[ ]:

```
'am Sin'
```

In [ ]:

```
#Everything
name[:]
```

Out[ ]:

```
'Shivam Singh'
```

In [ ]:

```
# Grab everything, but go in step sizes of 2
name[::2]
```

Out[ ]:

```
'Sia ig'
```

If you want to print a reverse of the given string name, then use

In [ ]:

```
name[::-1]
```

Out[ ]:

```
'hgniS mavihS'
```

String Properties

Immutability is one the finest string property whichh is created once and the elements within it cannot be changed or replaced.

In [ ]:

```
s
```

Out[ ]:

```
'Hello Python'
```

In [ ]:

```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-976942677f11> in <module>
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

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

Notice how the error tells us directly what we can't do, change the item assignment!

Something we can do is concatenate strings!

In [ ]:

```
s
```

Out[ ]:

```
'Hello Python'
```

In [ ]:

```
# Concatenate strings!
s + ' concatenate me!'
```

Out[ ]:

```
'Hello Python concatenate me!'
```

In [ ]:

```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

In [ ]:

```
print(s)
```

```
Hello Python concatenate me!
```

In [ ]:

```
s
```

Out[ ]:

```
'Hello Python concatenate me!'
```

We can use the multiplication symbol to create repetition!

In [ ]:

```
letter = 's'
```

In [ ]:

```
letter*10
```

Out[ ]:

```
'ssssssssss'
```

Basic Built-in String methods

In Python, Objects have built-in methods which means these methods are functions present inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

In [ ]:

```
str1 = "Aatmanirbahr Bharat"
```

In [ ]:

```
len(str1)
```

Out[ ]:

```
19
```

In [ ]:

```
# Upper Case a string
```

```
str1.upper()
```

Out[ ]:

```
'AATMANIRBAHR BHARAT'
```

In [ ]:

```
# Lower case
str1.lower()
```

Out[ ]:

```
'aatmanirbahr bharat'
```

In [ ]:

```
# Split a string by blank space (this is the default)
str1.split()
```

Out[ ]:

```
['Aatmanirbahr', 'Bharat']
```

In [ ]:

```
# Split by a specific element (doesn't include the element that was split on)
str1.split('B')
```

Out[ ]:

```
['Aatmanirbahr ', 'harat']
```

Location and Counting

In [ ]:

```
# count method
# The count method returns the number of occurrences of the substring substr in string st
r1.

str1.count('a')
```

Out[ ]:

```
5
```

In [ ]:

```
str1.count('a',5,19)
```

Out[ ]:

```
3
```

In many situations, we need to find the index of the substring in the given string. The find() method can do the task.

In [ ]:

```
str2 = "peace begins with a smile"
str2.find("with")
```

Out[ ]:

```
13
```

In [ ]:

```
# captalize method
# This method capitalizes the first letter of the returned string.
```

```
name = "shivam singh"
name.capitalize()
```

Out[ ]:

```
'Shivam singh'
```

If you want to convert the first character of every word of the string in uppercase, you can use the title() method.

In [ ]:

```
name.title()
```

Out[ ]:

```
'Shivam Singh'
```

Print Formatting

Print Formatting ".format()" method is used to add formatted objects to the printed string statements.

Let's see an example to clearly understand the concept.

In [ ]:

```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

Out[ ]:

```
'Insert another string with curly brackets: The inserted string'
```

String Boolean methods

Sometime we are interested in strings which are ends with particular substring. For this we use string method endswith().

In [ ]:

```
quote = "Life should be great rather than long"
quote.endswith("ng")
```

Out[ ]:

```
True
```

In [ ]:

```
quote.endswith("er")
```

Out[ ]:

```
False
```

In [ ]:

```
quote.endswith("er",0,27)
```

Out[ ]:

```
True
```

The next method is startswith(), which works the same way as the previous method, just check the condition from the beginning.

In [ ]:

```
quote.startswith("Li")
```

Out[ ]:

```
True
```

In [ ]:

```
quote.startswith("be")
```

Out[ ]:

```
False
```

In [ ]:

```
quote.startswith("be", 12, 16)
```

Out[ ]:

```
True
```

In [ ]:

```
a = "Hello"
```

isalnum() will return "True" if all characters in a are alphanumeric.

In [ ]:

```
a.isalnum()
```

Out[ ]:

```
True
```

In [ ]:

```
a1 = 'hello123@'
a1.isalnum()
```

Out[ ]:

```
False
```

isalpha() wil return "True" if all characters in a are alphabetic.

In [ ]:

```
a.isalpha()
```

Out[ ]:

```
True
```

In [ ]:

```
a2 = 'hello123'
a2.isalpha()
```

Out[ ]:

```
False
```

isdigit() method returns True if the string contains only digits.

In [ ]:

```
d1 = '12345'
d1.isdigit()
```

Out[ ]:

```
True
```

In [ ]:

```
d2 = '12345h'
d2.isdigit()
```

Out[ ]:

```
False
```

islower() method returns True if the string contains all lowercase characters.

In [ ]:

```
l1 = 'hello'
l1.islower()
```

Out[ ]:

```
True
```

In [ ]:

```
l2 = 'Hello'
l2.islower()
```

Out[ ]:

```
False
```

isupper() method returns True if the string contains all uppercase characters.

In [ ]:

```
u1 = 'HELLO'
u1.isupper()
```

Out[ ]:

```
True
```

In [ ]:

```
u2 = 'Hello'
u2.isupper()
```

Out[ ]:

```
False
```

isspace() will return "True" if all characters are whitespace.

In [ ]:

```
sp1 = "hello "
sp1.isspace()
```

Out[ ]:

```
False
```

In [ ]:

```
sp2 = " "
sp2.isspace()
```

Out[ ]:

```
True
```

istitle() will return "True" if String is a title cased string and there is at least one character in String, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return

False otherwise.

In [ ]:

```
n1 = "Shivam singh"
n1.istitle()
```

Out[ ]:

```
False
```

In [ ]:

```
n2 = "Shivam Singh"
n2.istitle()
```

Out[ ]:

```
True
```

String functions

So far you have seen string methods. Let's see built-in functions of sequences and what values they would return when the string is passed as an argument. At the beginning of the notebook, we have already discussed the len() function.

The min() function returns the min character from string str1 according to the ASCII value

In [ ]:

```
str1 = "Life should be great rather than long"
min(str1)
```

Out[ ]:

```
' '
```

In [ ]:

```
str2 = "hello!"
min(str2)
```

Out[ ]:

```
'!'
```

In [ ]:

```
str1 = "Life should be great rather than long"
max(str1)
```

Out[ ]:

```
'u'
```

In many situations, we might need to convert integers or floats into a string. In order to do this conversion, the str() function is used.

In [ ]:

```
a = 123
type(a)
```

Out[ ]:

```
int
```

In [ ]:

```
str(a)
```

Out[ ]:

```
'123'
```

Built-in Reg. Expressions

In Strings, there are some built-in methods which is similar to regular expression operations.

- **Split() function is used to split the string at a certain element and return a list of the result.**
- **Partition is used to return a tuple that includes the separator (the first occurrence), the first half and the end half.**

In [ ]:

```
s = 'Hello'
```

In [ ]:

```
s.split('e')
```

Out[ ]:

```
['H', 'llo']
```

In [ ]:

```
s.partition('e')
```

Out[ ]:

```
('H', 'e', 'llo')
```

Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they are being used.

Integers

You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

In [ ]:

```
2 + 3
```

Out[ ]:

```
5
```

In [ ]:

```
9 - 4
```

Out[ ]:

```
5
```

In [ ]:

```
4 * 3
```

Out[ ]:

```
12
```

In [ ]:

```
6 / 4
```

Out[ ]:

```
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents.

In [ ]:

```
4 ** 2
```

Out[ ]:

```
16
```

In [ ]:

```
3 ** 3
```

Out[ ]:

```
27
```

In [ ]:

```
2 ** 6
```

Out[ ]:

```
64
```

Floats

In [ ]:

```
0.1 + 0.1
```

Out[ ]:

```
0.2
```

In [ ]:

```
0.1 * 2
```

Out[ ]:

```
0.2
```

Complex Number

In [ ]:

```
c = 5 + 6j
```

In [ ]:

```
c.real
```

Out[ ]:

```
5.0
```

In [ ]:

```
c.imag
```

Out[ ]:

```
6.0
```

Floor division method

In [ ]:

```
6 // 4
```

Out[ ]:

```
1
```

In [ ]:

```
8 //3
```

Out[ ]:

```
2
```

Avoiding Type Errors with the str() Function

In [ ]:

```
age = 23
message = "Happy " + age + "rd Birthday!"

print(message)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-107-4d54be8cf23f> in <module>
      1 age = 23
----> 2 message = "Happy " + age + "rd Birthday!"
      3 
      4 print(message)

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

To avoid the error in above code we just need to convert the age into string by using the str() function.

In [ ]:

```
age = 23
message = "Happy " + str(age) + "rd Birthday!"

print(message)
```

```
Happy 23rd Birthday!
```

The Zen of Python

The Zen of Python is a collection of 19 "guiding principles" for writing computer programs that influence the design of the Python programming language. Software engineer Tim Peters wrote this set of principles and posted it on the Python mailing list in 1999. Peters's list left open a 20th principle "for Guido to fill in", referring to Guido van Rossum, the original author of the Python language. The vacancy for a 20th principle has not been filled.

Peters's Zen of Python was included as entry number 20 in the language's official Python Enhancement Proposals, which was released into the public domain. It is also included as an Easter egg in the Python interpreter, which can be displayed by entering `import this`.

In [ ]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Enumerating a String

In Python, the string is an array, and hence you can loop over it. If you pass a string to enumerate(), the output will show you the index and value for each character of the string.

In [1]:

```
my_str = "hello this is python batch"
for i in enumerate(my_str):
  print(i)
```

```
(0, 'h')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')
(5, ' ')
(6, 't')
(7, 'h')
(8, 'i')
(9, 's')
(10, ' ')
(11, 'i')
(12, 's')
(13, ' ')
(14, 'p')
(15, 'y')
(16, 't')
(17, 'h')
(18, 'o')
(19, 'n')
(20, ' ')
(21, 'b')
(22, 'a')
(23, 't')
(24, 'c')
(25, 'h')
```