**Strings**

Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

1.) Creating Strings

2.) Printing Strings

3.) String Indexing and Slicing

4.) String Properties

5.) String Methods

6.) Print Formatting

**Creating a String**

To create a string in Python you need to use either single quotes or double quotes. For example:

s **=** 'ASha' ,"",'''ASha''',"""Asha"""

s[0]

​

print("%s",s)

print("%r",s)

​

​

​

​

name **=** "Asha"

print("My name is %r" **%**name) \_\_repr\_\_()

print("My name is %s" **%**name) \_\_str\_\_()

My name is 'Asha'

My name is Asha

name **=** "Ram"

*# print(name)*

print(name)

name

​

Ram

Out[7]:

'Ram'

​

st **=** "Hello \t World"

c **=** r"C:\programs\newfolder"

print(st) *#str()*

print(c) *#repr() rawstring*

st

Hello World

C:\programs\newfolder

Out[29]:

'Hello \t World'

st **=** "Asha\tTalari"

print(st)

st

​

path **=** r"c:\newfolder\python"

​

​

st **=**"cat mat map man"

​

r"m\w\w"

​

r'select \* from table where name = 'ASha's'I'am'

​

​

st **=** r"Asha\tTalari"

print(st)

​

*# str() -- %s*

*# repr() -- %r - rawstring*

​

Asha Talari

Asha\tTalari

*# Single word*

'hello'

s **=** 'hello'

*# print(%rs) -- str()*

​

​

​

s **=** "Hello world"

print(s) str() **%**s **-** readable

s repr() **%**r **-** input code

​

Hello world

Out[4]:

'Hello world'

​

​

*# Entire phrase*

print("My name is Asha")

'This is also a string in nature'

My name is Asha

Out[9]:

'This is also a string in nature'

*# We can also use double quote*

"String built with double quotes"

Out[3]:

'String built with double quotes'

*# Be careful with quotes!*

"I'am musing single quotes, but this will create an error"

**File "<ipython-input-4-da9a34b3dc31>", line 2**

**' I'm using single quotes, but this will create an error'**

**^**

**SyntaxError:** invalid syntax

The reason for the error above is because the single quote in I'm stopped the string. You can use combinations of double and single quotes to get the complete statement.

print("Now I'm ready to \n use the single quotes inside a string!")

print(r"c:\new\asha")

Now I'm ready to

use the single quotes inside a string!

c:\new\asha

Now let's learn about printing strings!

**Printing a String**

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

*# We can simply declare a string*

*# print('Firstline')*

*# 'Hello World'*

​

s **=** 'Hello World'

print(s)

s

Hello World

Out[4]:

'Hello World'

​

​

*# Note that we can't output multiple strings this way*

print('Hello World 1') **%**s

'Hello World 2' **%**r

Hello World 1

Out[2]:

'Hello World 2'

We can use a print statement to print a string.

print('Hello World 1')

print('Hello World 2')

print(r"Use \t to print a new line")

print("Use \t to print a new line")

print('\n')

print('See what I mean?')

​

​

Hello World 1

Hello World 2

Use \t to print a new line

Use to print a new line

See what I mean?

print(r"Use \t to print a new line")

Use \t to print a new line

s **=** 'Moe\'s' *#use escape characters to print*

print(s)

s

​

Moe's

Out[3]:

"Moe's"

s **=**r'\nMoe\'s'

print(s)

\nMoe\'s

**String Basics**

We can also use a function called len() to check the length of a string!

len('Hello World')

sv**=**"Helloworld"

print(sv[0])

print(type(sv))

H

<class 'str'>

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

**String Indexing**

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets [] after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called s and then walk through a few examples of indexing.

*# Assign s as a string*

s **=** 'Hello World'

​

​

s[0]

​

s[0]**=**'h'

s[1]**=**'e'

..

s[10]**=**'d'

​

​

​

​

s[0]

h **-** 0

e **-** 1

l **-**2

​

d **-** **-**1

l **-** **-**2

r **-** **-**3

​

​

s[0]

s[**-**1]

​

*# incremental*

*# 0,1,2,3,4..*

*# decremental - negative indexing*

*# -1,-2,-3,-4....*

​

*# print(s[-2])*

*# print(s[0])*

Out[5]:

'd'

​

st **=**"Hello World"

​

​

print(st[0])

print(st[10])

​

print(st[**-**1])

print(st[**-**11])

H

d

d

H

st **=** "Hello World"

print(st[0],st[1],st[2],st[10]) *#positive indexing*

*#negative indexing*

print(st[**-**1],st[**-**9],st[**-**10])

​

H e l d

d l e

​

*#Check*

s**=** "Hello World"

*#positive indexing*

print(s[1])

​

*#negative indexing*

*# s[-1] ='d'*

*# s[-2]='l'*

*# s[-3]='r'*

​

print(s[**-**1])

print(s[**-**4])

​

​

e

d

o

​

Let's start indexing!

*# Show first element (in this case a letter)*

print(s[0])

print(s[1])

H

e

s[1]

Out[14]:

'e'

s[2]

Out[15]:

'l'

String slices We can use a : to perform *slicing* which grabs everything up to a designated point. For example:

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

The operator [n:m] returns the part of the string from the n-eth'' character to them-eth'' character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

An empty string contains no characters and has length 0, but other than that, it is the same as any other string. Exercise 2 Given that fruit is a string, what does fruit[:] mean?

s **=** 'Monty Python'

​

s[0]

s[::]

s[6:10:1]

s[0:5]

s[startindex:endindex:stepsize]

​

s[0:12:1]

s[::]

​

s[startindex:endindex:stepsize]

​

s[startindex:endindex:stepsize]

*#[startindex:endindex:stepsize]*

​

print(s[**-**3:8:**-**1])

print(s[3:7:1])

​

*#s[-3:8:-1]*

​

​

h

​

​

s **=** 'Monty Python'

*# print(s[0:5:-1])*

print(s[5:0:**-**1])

ytno

st **=** 'Monty Python'

print(st[2:7:1])

nty P

s **=** "Monty Python"

print(s[**-**2:**-**5:**-**1])

oht

fruit **=** 'banana'

print(fruit[:3])

print(fruit[3::])

ban

ana

fruit **=** 'banana'

fruit[3:3:1]

Out[10]:

''

*# Grab everything past the first term all the way to the length of s which is len(s)*

s[1:]

Out[11]:

'onty Python'

*# Note that there is no change to the original s*

print(s)

Monty Python

*# Grab everything UP TO the 3rd index*

s[:3]

Out[12]:

'Mon'

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

*#Everything*

s[1:3:1]

Out[16]:

'on'

We can also use negative indexing to go backwards. The negative sign, -, means that we start at the third-to-last letter, and the colon means that we go to the end.

*# Last letter (one index behind 0 so it loops back around) ,backwards starts from -1,-2,-3*

s **=** "Monty Python"

print(s[3:**-**10:**-**1])

print(s)

​

​

t

Monty Python

*# Grab everything but the last letter , need to omit the last letter as it is -1*

s[:**-**1]

*# s[0:-1:1] - Monty Python*

Out[8]:

'Monty Pytho'

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

*# Grab everything, but go in steps size of 1*

s[::1]

Out[22]:

'Hello World'

*# Grab everything, but go in step sizes of 2*

s[::2]

Out[23]:

'HloWrd'

*# We can use this to print a string backwards*

*# s[::-1]*

s **=** "Hello World"

print(s[5:**-**8:**-**1])

print(s[:])

​

o

Hello World

**String Properties**

Strings are immutable It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed or replaced. For example:

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

greeting **=** 'Hello, world!'

​

​

​

greeting[0] **=** 'J'

greeting **=** "Hello"

​

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-43-446f48905c7e>** in <module>

1 greeting **=** **'Hello, world!'**

**----> 2** greeting**[0]** **=** **'J'**

3

4

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



The ``object'' in this case is the string and the ``item'' is the character you tried to assign.

For now, an object is the same thing as a value, but we will refine that definition later.

An item is one of the values in a sequence.

​

The reason for the error is that strings are immutable,

which means you can't change an existing string.

The best you can do is create a new string that is a variation on the original:

greeting **=** 'Hello, world!'

print(id(greeting))

greeting **=**"Jello World!"

print(id(greeting))

new\_greeting **=** 'J' **+** greeting[1:]

print(new\_greeting)

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

​

107413160

104847144

Jello World!

s

Out[25]:

'Hello World'

*# Let's try to change the first letter to 'x'*

s[0] **=** 'x'

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-26-976942677f11>** 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!

s

Out[27]:

'Hello World'

*# Concatenate strings!*

s **+** ' concatenate me!'

Out[28]:

'Hello World concatenate me!'

*# We can reassign s completely though!*

s **=** s **+** ' concatenate me!'

print(s)

Hello World concatenate me!

s

Out[31]:

'Hello World concatenate me!'

We can use the multiplication symbol to create repetition!

letter **=** 'z'

*# letter\*10*

print("Asha")

​

print("Hello",end**=**' ')

​

print('\*' **\***10) *#,end='')*

print("World")

print("Hello",end **=** " ")

print("First")

Asha

Hello \*\*\*\*\*\*\*\*\*\*

World

Hello First

lst **=** [10,20,30] *#when \* is used to repeated twice*

print(lst**\***2)

​

lst **=** [10,20,30] *#when \* is used to multiple indiviual values through comprehension*

lst1 **=** [x**\***2 **for** x **in** lst]

print(lst1)

​

​

**def** func(lst): *#through normal functional call*

lste **=** []

**for** x **in** lst:

lste.append(x**\***2)

**return** lste

​

print(func(lst))

​

​

​

[10, 20, 30, 10, 20, 30]

[20, 40, 60]

[20, 40, 60]

stuff **=** "Asha"

print(type(stuff))

​

print(dir(stuff))

<class 'str'>

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_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', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

**Basic Built-in String methods**

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

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

**string methods**

Strings are an example of Python objects. An object contains both data (the actual string itself) as well as methods, which are effectively functions which that are built into the object and are available to any instance of the object.

Python has a function called dir that lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

stuff = 'Hello world'

type(stuff)

<type 'str'>

dir(stuff)

['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']

help(str.capitalize)

Help on method\_descriptor:

capitalize(...) S.capitalize() -> string

Return a copy of the string S with only its first character

capitalized.

While the dir function lists the methods, and you can use help to get some simple documentation on a method,

a better source of documentation for string methods would be docs.python.org/library/string.html.

Calling a method is similar to calling a function---it takes arguments and returns a value---but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method upper takes a string and returns a new string with all uppercase letters:

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

word = 'banana'

new\_word = word.upper()

print (new\_word)

BANANA

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

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

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

word = 'banana'

index = word.find('a')

print(index)

1

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

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

print(word.find('na'))

2

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

print(word.find('na', 3))

4

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the strip method:

line = ' Here we go '

line.strip()

'Here we go'

Some methods such as startswith return boolean values.

line = 'Please have a nice day'

line.startswith('Please')

True

line.startswith('p')

False

You will note that startswith requires case to match so sometimes we take a line and map it all to lowercase before we do any checking using the lower method.

line = 'Please have a nice day'

line.startswith('p')

False

line.lower()

'please have a nice day'

line.lower().startswith('p')

True

In the last example, the method lower is called and then we use startswith to check to see if the resulting lowercase string starts with the letter ``p''. As long as we are careful with the order, we can make multiple method calls in a single expression.

stuff **=** 'Hello world'

​

print(type(stuff))

​

*# dir(stuff)*

<class 'str'>

help(str.capitalize)

Help on method\_descriptor:

capitalize(self, /)

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower

case.

word **=** 'banana'

​

​

new\_word **=** word.upper()

​

print(word)

​

print (new\_word)

banana

BANANA

print(word.find('na'))

2

print(word.find('na', 3))

4

line **=** ' Here we go '

line.strip()

Out[28]:

'Here we go'

line **=** 'Please have a nice day'

​

line.startswith('Please')

Out[30]:

True

line.startswith('p')

Out[31]:

False

line.lower()

line.lower().startswith('p')

Out[32]:

True

​

**Parsing strings**

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

From stephen.marquard@ uct.ac.za Sat Jan 5 09:14:16 2008

And we wanted to pull out only the second half of the address (i.e. uct.ac.za) from each line. We can do this by using the find method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space after the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

data = 'From [stephen.marquard@uct.ac.za](mailto:stephen.marquard@uct.ac.za) Sat Jan 5 09:14:16 2008'

atpos = data.find('@')

print(atpos)

21

sppos = data.find(' ',atpos)

print(sppos)

31

host = data[atpos+1:sppos]

print(host)

uct.ac.za

We use a version of the find method which allows us to specify a position in the string where we want find to start looking. When we slice, we extract the characters from ``one beyond the at-sign through up to but not including the space character''.

data **=** 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'

​

atpos **=** data.find('@')

​

print(atpos)

21

sppos **=** data.find(' ',atpos)

​

print(sppos)

31

host **=** data[atpos**+**1:sppos] *#range*

​

print(host)

uct.ac.za

a **=** "Asha"

print(dir(a))

help(a.capi)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_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', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

s

Out[36]:

'Hello World'

​

*# Upper Case a string*

s.upper()

Out[35]:

'HELLO WORLD CONCATENATE ME!'

​

print(help(s.title))

Help on built-in function title:

title() method of builtins.str instance

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining

cased characters have lower case.

None

*# Lower case*

s.lower()

Out[36]:

'hello world concatenate me!'

*# Split a string by blank space (this is the default)*

​

s **=** "Hello World"

s.split()

Out[34]:

['Hello', 'World']

*# Split by a specific element (doesn't include the element that was split on)*

s.split('W')

Out[38]:

['Hello ', 'orld concatenate me!']

There are many more methods than the ones covered here. Visit the Advanced String section to find out more!

**Print Formatting**

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

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

Out[39]:

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

Type *Markdown* and LaTeX: 𝛼2α2

Type *Markdown* and LaTeX: 𝛼2α2

**Format operator**

The format operator, % allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the format string, which contains one or more format sequences that specify how the second operand is formatted. The result is a string.

For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for ``decimal''):

camels = 42

print('%d' % camels)

'42' The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

camels = 42

print('I have spotted %d camels.' % camels)

'I have spotted 42 camels.'

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number (don't ask why), and '%s' to format a string:

print('In %d years I have spotted %g %s.' % (3, 0.1, 'camels'))

'In 3 years I have spotted 0.1 camels.'

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

print('%d %d %d' % (1, 2))

TypeError: not enough arguments for format string

print('%d' % 'dollars')

TypeError: illegal argument type for built-in operation In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use.

camels **=** 42

print('Camel value is %d' **%**camels)

​

Camel value is 42

print(type(camels))

<class 'int'>

print('In %d years I have spotted %g %s.' **%** (3, 0.1, 'camels'))

In 3 years I have spotted 0.1 camels.

print('%d %d %d' **%** (1, 2))

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-18-347a2ce4f617>** in <module>

**----> 1** print**('%d %d %d'** **%** **(1,** **2))**

**TypeError**: not enough arguments for format string

print('%d' **%** 'dollars')

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-19-104f39149fcb>** in <module>

**----> 1** print**('%d'** **%** **'dollars')**

**TypeError**: %d format: a number is required, not str

*#Formatting of strings in detail in next presentation*

**Traversal through a string with a loop**

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a while loop:

*#This loop traverses the string and displays each letter on a line by itself.*

*#The loop condition is index < len(fruit), so when index is equal to the length of the string,*

*#the condition is false, and the body of the loop is not executed.*

*#The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.*

​

fruit **=**'banana'

index **=** 0

**while** index **<** len(fruit):

letter **=** fruit[index]

print(letter)

index **=** index **+** 1

b

a

n

a

n

a

*#Another way to write a traversal is with a for loop:*

fruit **=** 'Banana'

print(fruit[::**-**1])

**for** char **in** fruit[::**-**1]:

print(char)

*#Each time through the loop, the next character in the string is assigned to the variable char.*

*#The loop continues until no characters are left.*

0

ananaB

a

n

a

n

a

B

*#Looping and counting*

*#The following program counts the number of times the letter a appears in a string:*

​

word **=** 'banana'

count **=** 0

**for** letter **in** word:

**if** letter **==** 'a':

count **=** count **+** 1

print(count)

*#This program demonstrates another pattern of computation called a counter.*

*#The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result---the total number of a's.*

​

3

**The in operator**

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

'a' **in** 'banana'

Out[48]:

True

'seed' **in** 'banana'

Out[49]:

False

**String comparison**

The comparison operators work on strings. To see if two strings are equal:

**if** word **==** 'banana':

print('All right, bananas.')

​

*#Other comparison operations are useful for putting words in alphabetical order:*

​

**if** word **<** 'banana':

print('Your word,' **+** word **+** ', comes before banana.')

**elif** word **>** 'banana':

print('Your word,' **+** word **+** ', comes after banana.')

**else**:

print('All right, bananas.')

*#Python does not handle uppercase and lowercase letters the same way that people do.*

*#All the uppercase letters come before all the lowercase letters, so:*

​

*#Your word, Pineapple, comes before banana.*

*#A common way to address this problem is to convert strings to a standard format,*

*#such as all lowercase, before performing the comparison.*

*#Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.*

​

All right, bananas.

index **=** 0

index1 **=** 0

name **=** "Monisha"

print(len(name))

**while**(index **>** len(name)):

index **=** index **+** 1

index1 **=** index1**-**1

print(index)

print(name[index])

7

​

10

11

12

13

14

15

16

17

18

19

​

**Advanced Strings**

String objects have a variety of methods we can use to save time and add functionality.

s **=** 'hello world'

**Changing case**

We can use methods to capitalize the first word of a string, or change the case of the entire string.

*# Capitalize first word in string*

s.capitalize()

Out[2]:

'Hello world'

s.upper()

Out[3]:

'HELLO WORLD'

s.lower()

Out[4]:

'hello world'

Remember, strings are immutable. None of the above methods change the string in place, they only return modified copies of the original string.

s

Out[5]:

'hello world'

To change a string requires reassignment:

s **=** s.upper()

s

Out[6]:

'HELLO WORLD'

s **=** s.lower()

s

Out[7]:

'hello world'

**Location and Counting**

s.count('o') *# returns the number of occurrences, without overlap*

Out[9]:

2

s.find('o') *# returns the starting index position of the first occurence*

Out[10]:

4

**Formatting**

The center() method allows you to place your string 'centered' between a provided string with a certain length. Personally, I've never actually used this in code as it seems pretty esoteric...

s.center(20,'z')

Out[11]:

'zzzzhello worldzzzzz'

The expandtabs() method will expand tab notations \t into spaces:

'hello\thi'.expandtabs()

Out[12]:

'hello hi'

**is check methods**

These various methods below check if the string is some case. Let's explore them:

s **=** 'hello12'

isalnum() will return True if all characters in **s** are alphanumeric

s.isalnum()

Out[14]:

True

isalpha() will return True if all characters in **s** are alphabetic

s.isalpha()

Out[15]:

True

islower() will return True if all cased characters in **s** are lowercase and there is at least one cased character in **s**, False otherwise.

s.islower()

Out[16]:

True

isspace() will return True if all characters in **s** are whitespace.

s.isspace()

Out[17]:

False

istitle() will return True if **s** is a title cased string and there is at least one character in **s**, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns False otherwise.

s.istitle()

Out[18]:

False

isupper() will return True if all cased characters in **s** are uppercase and there is at least one cased character in **s**, False otherwise.

s.isupper()

Out[19]:

False

Another method is endswith() which is essentially the same as a boolean check on s[-1]

s.endswith('o')

Out[20]:

True

**Built-in Reg. Expressions**

Strings have some built-in methods that can resemble regular expression operations. We can use split() to split the string at a certain element and return a list of the results. We can use partition() to return a tuple that includes the first occurrence of the separator sandwiched between the first half and the end half.

s **=** "Hello World to the Python"

*# s = "Jello"*

y**=**s.split()

print(s.split())

​

['Hello', 'World', 'to', 'the', 'Python']

s.partition('l')

Out[4]:

('He', 'l', 'lo')

**Docstrings**

Actually python supports only single line comments , multiline comments are not availble in python , Single and double triples quotes are not actually comments instead they are regular strings with an extension that they can span over multiple lines it means memory will be allocated internally for these strings If theese strings are not assigned to any varaible then it will be removed by the Garbage collector , that is reason it will be used as comments And as per python experts it is not recomended to use these multiline comments as it occupies memory

If these triple quote strings are written as first line in any program like in class, function or module then it can be called as documentation strings or docstrings. These docstrings are useful to create API documentation file from python program.

To test this we can write simple program in IDLE and compile to create API documentation To create the documentation file we use pydoc module and the command is

c/> python -m pydoc -w nameofthe file

So here both the file gets compiled and executed along with creating the html file , which can be used for documentation purpose

**!**python **-**m pydoc **-**w Testing

50

wrote Testing.html

**!**python **-**m pydoc **-**w Testing *#while creating the API doc file we should not specify the extension*

50

No Python documentation found for 'Testing.py'.

Use help() to get the interactive help utility.

Use help(str) for help on the str class.

**!**dir

Volume in drive C is Windows

Volume Serial Number is 26EA-BC3B

Directory of C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\1 - Python Introduction and Python Objects

08/28/2020 03:19 PM <DIR> .

08/28/2020 03:19 PM <DIR> ..

07/21/2020 06:14 PM <DIR> .ipynb\_checkpoints

08/26/2020 12:22 PM 6,430 01-Advanced Numbers.ipynb

08/26/2020 04:16 PM 28,583 01-Numbers.ipynb

08/28/2020 02:11 PM 12,692 02-Advanced Strings.ipynb

08/26/2020 04:47 PM 20,382 02-Variable Assignment.ipynb

08/27/2020 04:14 PM 69,472 03-Strings.ipynb

08/28/2020 02:05 PM 18,427 04-Print Formatting with Strings.ipynb

08/26/2020 04:43 PM 35 annotate.py

07/09/2020 10:09 PM <DIR> Assessments

08/26/2020 05:16 PM 12,816 Datatypes - Introduction.ipynb

04/01/2020 03:35 PM <DIR> owndir1

08/24/2020 04:05 PM 15,995 Python - Introduction.ipynb

07/14/2020 09:23 PM 5,443 Python - Syntax-Semanticerrorcheck.ipynb

08/24/2020 03:28 PM 9,032 Python Installation.ipynb

08/28/2020 02:07 PM 1,516 Testing.html

08/28/2020 03:19 PM 114 Testing.py

08/28/2020 02:07 PM <DIR> \_\_pycache\_\_

13 File(s) 200,937 bytes

6 Dir(s) 403,164,459,008 bytes free