Chapter 8: Python Strings

Contents

Red light: No AI

Time required: 120 minutes

DRY

Don't **R**epeat **Y**ourself

String Basics

A string is a data type in Python that is a sequence of characters.

Creating a string: A string is created by enclosing text in quotes. You can use single quotes, ', or double quotes, ". A triple-quote is used for multi-line strings. Here are some examples:

```
s = "Hello"
t = 'Hello'
m = """
```

```
This is a long string that is
spread across two lines.
"""
```

Input: When we are getting text from the user, we use input. This is illustrated below:

```
num = int(input("Enter a whole number: "))
string = input("Enter a string: ")
```

Empty string: The empty string "" is the string equivalent of the number 0. It is a string with nothing in it. It is in the print statement’s optional argument, **sep=""**.

A String is a Sequence

A string is a sequence of characters. It is like a list or tuple. Each character has an index number starting with 0. You can access the characters one at a time with the bracket operator:

```
fruit = "banana"
letter = fruit[1]
print(letter)

a
```

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.

The expression in brackets is called an index. The index indicates which character in the sequence you want.

You might not get what you expect:

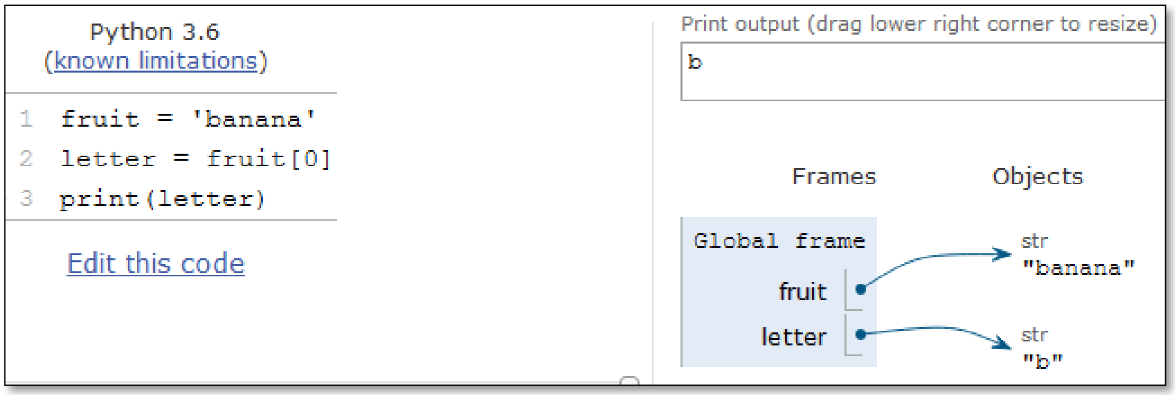
```
fruit = "banana"
letter = fruit[1]
print(letter)

a
```

For most people, the first letter of “banana” is “b”, not “a”. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
fruit = "banana"
letter = fruit[0]
print(letter)
```

```
b
```

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


Python 3.6
(known limitations)

```
fruit = 'banana'
letter = fruit[0]
print(letter)
```

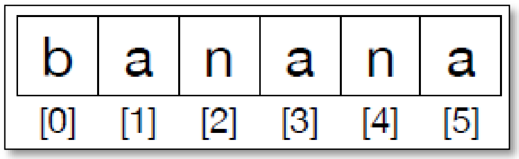
Edit this code

Print output (drag lower right corner to resize)

```
b
```

You can use any expression, including variables and operators, as an index. The value of the index must be an integer.

```
letter = fruit[1.5]
TypeError: string indices must be integers
```



Getting the Length of a String using len

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

```
fruit = "banana"
len(fruit)

6
```

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

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

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

```
fruit = "banana"
last = fruit[length - 1]
>>> print(last)

a
```

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

The empty string "" is the string equivalent of the number 0. It is a string with nothing in it.

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:

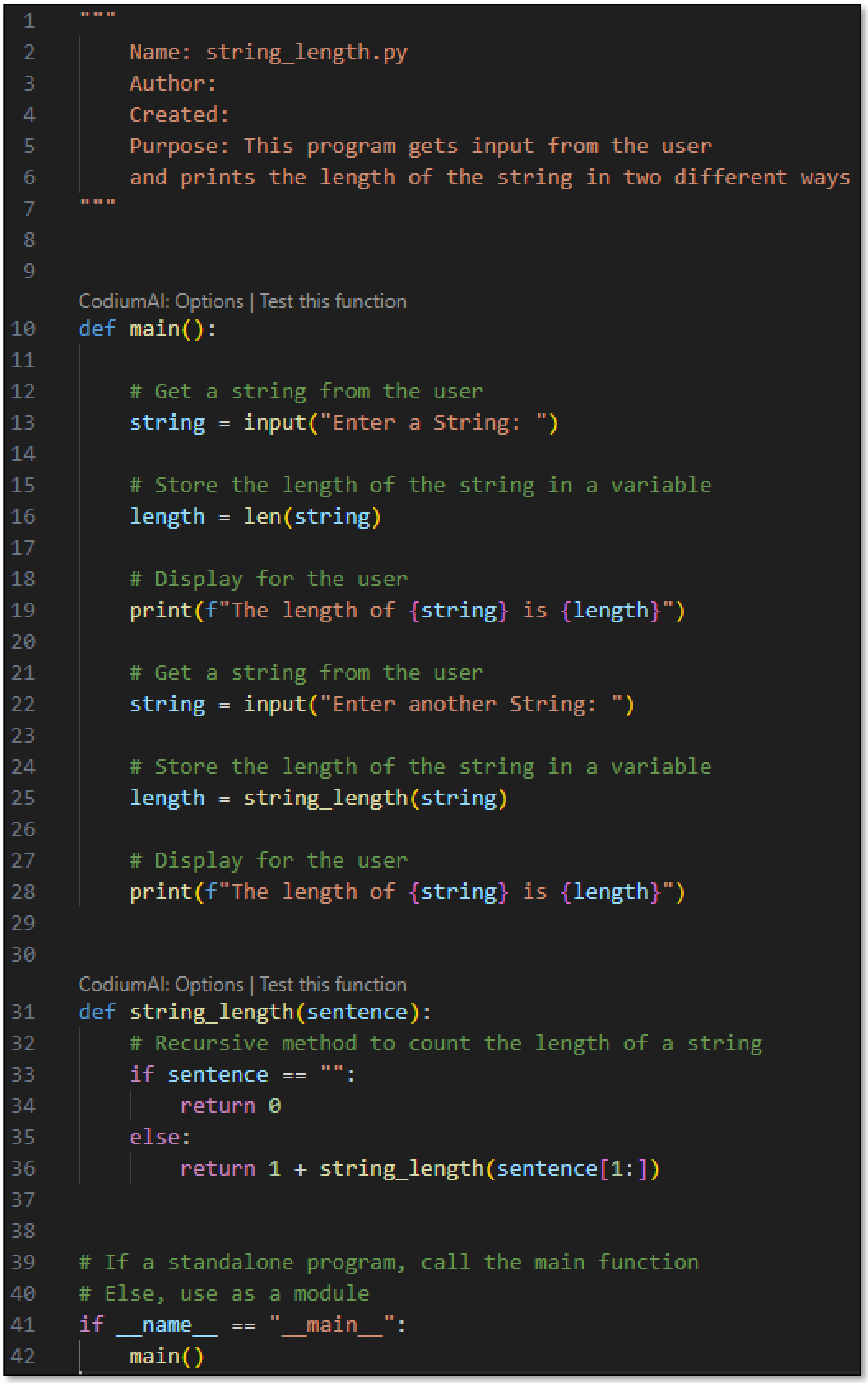
```
Your word, Pineapple, comes before banana.
```

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 the next time you must defend yourself against a man armed with a Pineapple.

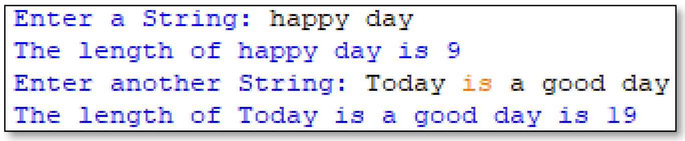
Tutorial 8.1: String Length

Create and test the following Python Program named **string_length.py**



```python
"""
    Name: string_length.py
    Author:
    Created:
    Purpose: This program gets input from the user
    and prints the length of the string in two different ways
"""


CodiumAI: Options | Test this function
def main():

    # Get a string from the user
    string = input("Enter a String: ")

    # Store the length of the string in a variable
    length = len(string)

    # Display for the user
    print(f"The length of {string} is {length}")

    # Get a string from the user
    string = input("Enter another String: ")

    # Store the length of the string in a variable
    length = string_length(string)

    # Display for the user
    print(f"The length of {string} is {length}")


CodiumAI: Options | Test this function
def string_length(sentence):
    # Recursive method to count the length of a string
    if sentence == "":
        return 0
    else:
        return 1 + string_length(sentence[1:])


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
Enter a String: happy day
The length of happy day is 9
Enter another String: Today is a good day
The length of Today is a good day is 19
```

Traversal through a String with a Loop

We often want to pick out individual characters from a string. Python uses square brackets to do this. The table below gives some examples of indexing the string s="Python".

Statement	Result	Description
s[0]	P	first character of s
s[1]	y	second character
s[-1]	n	of s last character
s[-2]	o	of s

- The first character of s is s[0], not s[1]. Remember that in programming, counting usually starts at 0, not 1.
- Negative indices count backward from the end of the string.

Many computations involve processing a string one character at a time. 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:

```
fruit = "banana"
index = 0
while index < len(fruit):
   letter = fruit[index]
   print(letter)
   index = index + 1
```

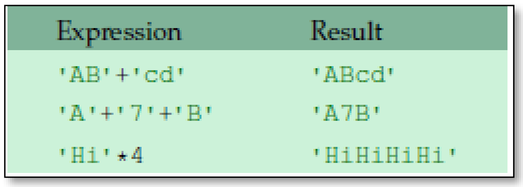
This loop traverses the string and displays each letter on a line by itself. The loop condition is **index < len(fruit).** When index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index **len(fruit)-1**, which is the last character in the string.

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

```
fruit = "banana"
for char in fruit:
   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.

The operators + and * can be used on strings. The + operator combines two strings. This operation is called concatenation. The * repeats a string a certain number of times.



Expression	Result
'AB'+'cd'	'ABcd'
'A'+'7'+'B'	'A7B'
'Hi'*4	'HiHiHiHi'

If we want to print a long row of dashes, we can do the following

```
print("-" * 75)
```

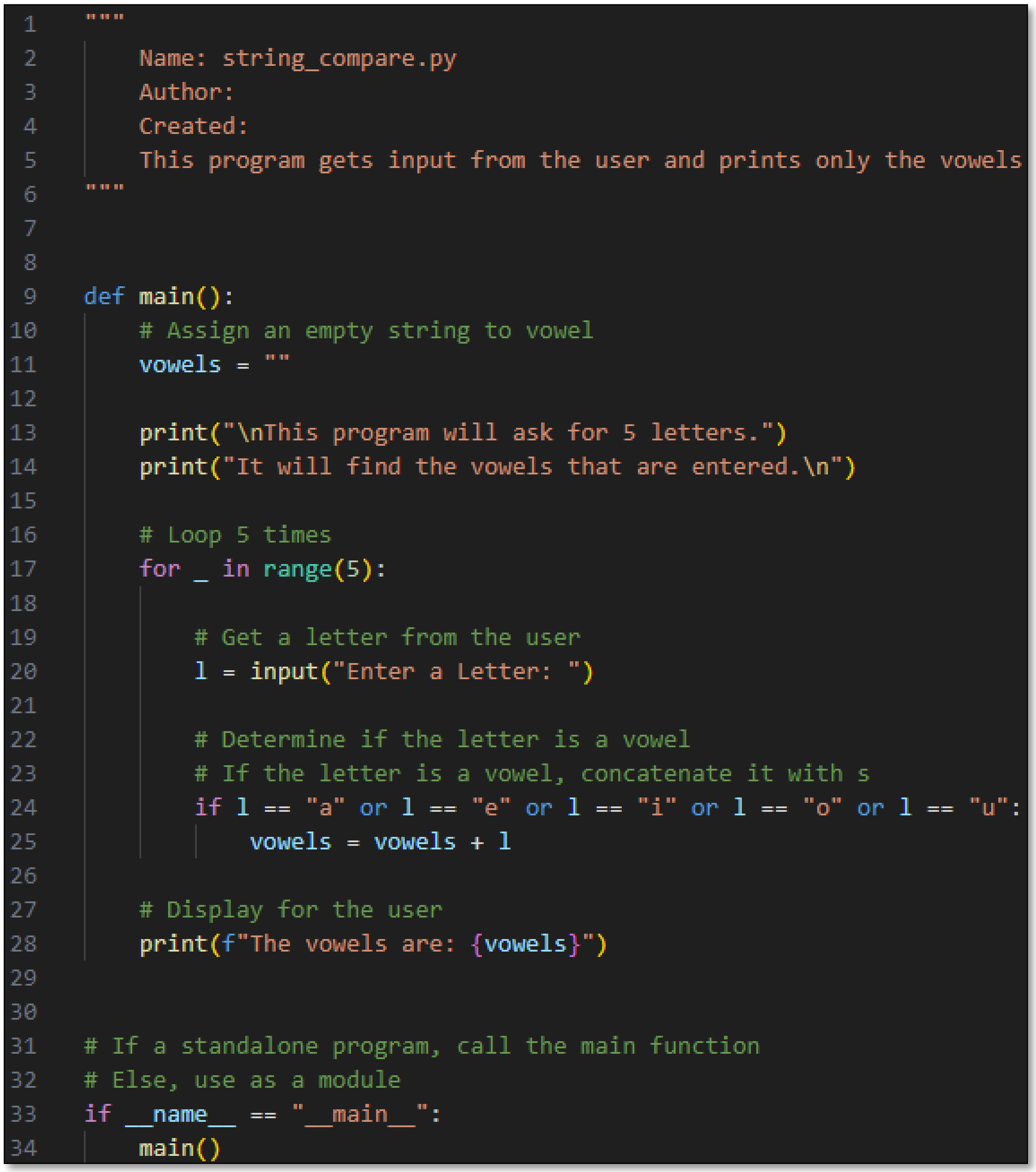
Strings can be compared for equality. The first line assigns the value 'a' to the string. The second line compares the value 'a' to the contents of the string.

```
t = "a"
t == "a"
```

Tutorial 8.2: String Compare

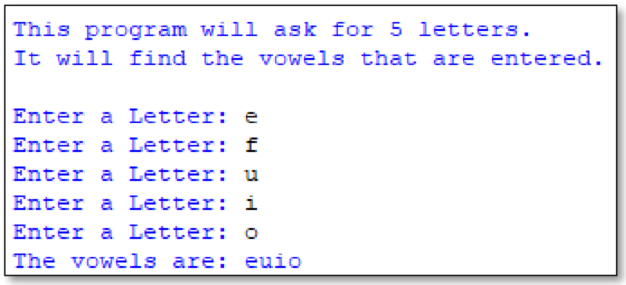
The example program repeatedly asks the user to enter a letter and builds up a string consisting of only the vowels that the user entered.

Create and test a Python program named **string_compare.py**



```python
"""
    Name: string_compare.py
    Author:
    Created:
    This program gets input from the user and prints only the vowels
"""


def main():
    # Assign an empty string to vowel
    vowels = ""

    print("\nThis program will ask for 5 letters.")
    print("It will find the vowels that are entered.\n")

    # Loop 5 times
    for _ in range(5):

        # Get a letter from the user
        l = input("Enter a Letter: ")

        # Determine if the letter is a vowel
        # If the letter is a vowel, concatenate it with s
        if l == "a" or l == "e" or l == "i" or l == "o" or l == "u":
            vowels = vowels + l

    # Display for the user
    print(f"The vowels are: {vowels}")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example program run:



```
This program will ask for 5 letters.
It will find the vowels that are entered.

Enter a Letter: e
Enter a Letter: f
Enter a Letter: u
Enter a Letter: i
Enter a Letter: o
The vowels are: euio
```

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:

```
>>> "i" in "banana"
True
>>> "seed" in "banana"
False
```

The in operator can be used to tell if a string contains something. For example:

```
string = "banana"
if "a" in string:
   print("Your string contains the letter a.")
```

You can combine in with the not operator to tell if a string does not contain something:

```
string = "banana"
if ';' not in string:
   print("Your string does not contain any semicolons. ")
```

In the previous section we had the long if condition:

```
if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
```

Using the **in** operator, we can replace that statement with the following:

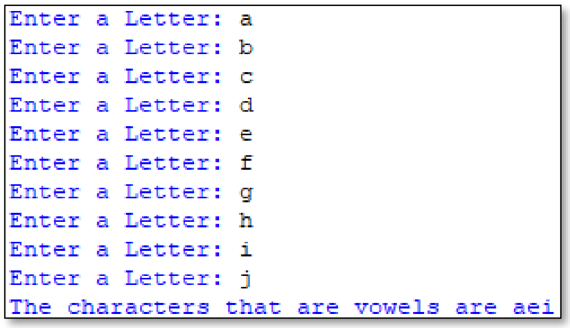
```
if t in "aeiou":
```

Tutorial 8.3: String Compare with in

Rewrite the **string_compare.py** program we wrote earlier and save it as **string_compare_in.py**

Use the in operator to find the vowels in a list of vowels as shown above.

Example program run:



```
Enter a Letter: a
Enter a Letter: b
Enter a Letter: c
Enter a Letter: d
Enter a Letter: e
Enter a Letter: f
Enter a Letter: g
Enter a Letter: h
Enter a Letter: i
Enter a Letter: j
The characters that are vowels are aei
```

Slicing

A slice is used to pick out part of a string. It behaves like a combination of indexing and the range function.

Below we have some examples with the string **s = "abcdefghij"**

```
index:    0 1 2 3 4 5 6 7 8 9
letters:  a b c d e f g h i j
```

Code	Result	Description
`s[2:5]`	cde	characters at indices 2, 3, 4
`s[ :5]`	abcde	first five characters
`s[5: ]`	fghij	characters from index 5 to the end
`s[-2: ]`	ij	last two characters
`s[ : ]`	abcdefghij	entire string
`s[1:7:2]`	bdf	characters from index 1 to 6, by twos
`s[ : :-1]`	jihgfedcba	a negative step reverses the string

The basic structure is

```
string name[starting location : ending location+1]
```

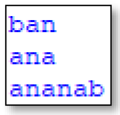
```
s = "Monty Python"
print(s[0:5])
print(s[6:12])

Monty
Python
```

Slices have the same quirk as the range function in that they do not include the ending location. For instance, in the example above, `s[2:5]` gives the characters in indices 2, 3, and 4, but not the character in index 5.

We can leave either the starting or ending locations blank. If we leave the starting location blank, it defaults to the start of the string. `s[:5]` gives the first five characters of s. If we leave the ending location blank, it defaults to the end of the string. So `s[5:]` will give all the characters from index 5 to the end. If we use negative indices, we can get the ending characters of the string. For instance, `s[-2:]` gives the last two characters.

```
# Filename: string_reverse_slice.py
fruit = "banana"
print(fruit[:3])
print(fruit[3:])
# Slice entire string, -1 reverses step
backwards = fruit[::-1]
print(backwards)
```

Example run:



```
ban
ana
ananab
```

There is an optional third argument, just like in the range statement, that can specify the step. For example, `s[1:7:2]` steps through the string by twos, selecting the characters at indices 1, 3, and 5 (but not 7, because of the aforementioned quirk). The most useful step is -1, which steps backward through the string, reversing the order of the characters. This is shown in the above example

Strings are Immutable

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

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

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

The reason for the error is that strings are immutable, which means 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! "
new_greeting = "J" + greeting[1:]
print(new_greeting)

Jello, world!
```

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

Changing Individual Characters of a String

Suppose we have a string called s and we want to change the character at index 4 of s to 'X'. It is tempting to try `s[4]='X'`, but unfortunately that will not work. Python strings are immutable, which means we can’t modify any part of them.

If we want to change a character of s, we build a new string from s and reassign it to s. Here is code that will change the character at index 4 to 'X':

```
s = s[:4] + 'X' + s[5:]
```

The idea of this is we take all the characters up to index 4, then X, and then all the characters after index 4.

Looping

Very often we will want to scan through a string one character at a time. A for loop like the one below can be used to do that. It loops through a string called s, printing the string, character by character, each on a separate line:

```
for i in range(len(s)):
    print (s[i])
```

In the range statement, we have `len(s)` that returns how long s is. So, if s were 5 characters long, this would be like having `range(5)` and the loop variable i would run from 0 to 4. This means that `s[i]` will run through the characters of s. This way of looping is useful if we need to keep track of our location in the string during the loop.

If we don’t need to keep track of our location, then there is a simpler type of loop we can use:

```
for character in string:
    print(character)
```

This loop will step through s, character by character, with c holding the current character. You can almost read this like an English sentence, “For every character c in s, print that character.”

The following program counts the number of times the letter “a” appears in a string:

```
count = 0
word = input('Enter a sentence: ')
letter = input('Enter a letter to count: ')
for ch in word:
   # Use the lower() method to compare
   # Upper and lower case entries
   if ch.lower() == letter.lower():
      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.

String Methods

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

Strings come with a ton of methods, functions that return information about the string or return a new string that is a modified version of the original. Here are some of the most useful ones:

Method	**Description**
`lower()`	returns a string with every letter of the original in lowercase
`upper()`	returns a string with every letter of the original in uppercase
`title()`	returns a string with the first letter of the original in uppercase
`replace(x, y)`	returns a string with every occurrence of x replaced by y

`count(x)`	counts the number of occurrences of x in the string
`index(x)`	returns the location of the first occurrence of x
`isalpha()`	returns True if every character of the string is a letter
`split()`	returns a string with every specified character (ch) removed
`capitalize()`	returns a string with the first letter capitalized, the rest of the string is lowercase.

Important note: One very important note about lower, upper, and replace is that they do not change the original string. If you want to change a string, s, to all lowercase, it is not enough to just use s.lower(). You need to do the following:

```
s = s.lower()
```

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.

```
word = "banana"
new_word = word.upper()
print(new_word)
Result: BANANA
```

Here are some examples of string methods in action:

Statement	**Description**
`print(s.count(""))`	prints the number of spaces in the string
`s = s.upper()`	changes the string to all caps
`s = s.replace("Hi","Hello")`	replaces each 'Hi' in s with 'Hello'
`print(s.index("a"))`	prints location of the first 'a' in s
`print(s.split())`	splits the string s at every space

isalpha

The isalpha method is used to tell if a character is a letter or not. It returns True if the character is a letter and False otherwise. When used with an entire string, it will only return True if every character of the string is a letter. Here is a simple example:

```
s = input("Enter  a  string")
if s[0].isalpha():
   print('Your string starts with a letter')
if not s.isalpha():
   print('Your string contains a non-letter.')
```

A note about index If you try to find the index of something that is not in a string, Python will raise an error. For instance, if s='abc' and you try s.index('z'), you will get an error. One way around this is to check first, like below:

```
if 'z' in s:
   location = s.index('z')
```

Find searches for the position of one string within another:

```
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.

The find method can find substrings as well as characters:

```
word.find('na')

2
```

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

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

4
```

strip()

Remove white space (spaces, tabs, or newlines) from the beginning and end of a string.

```
line = '    Here we go    '
print(line)
print(line.strip())
```

startswith()

returns boolean values.

```
line = 'Have a nice day'
print(line.startswith('Have'))

True

print(line.startswith('h'))

False
```

`startswith` requires case to match. Sometimes we take a line and map it all to lowercase before we do any checking using the lower method.

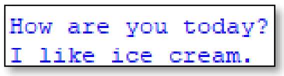
```
line = 'Have a nice day'
line.startswith('h')

False

line.lower()
# 'have a nice day'
line.lower().startswith('h')

True
```

In the last example, the method lower is called and then we use **startswith()** to see if the resulting lowercase string starts with the letter “h”. If we are careful with the order, we can make multiple method calls in a single expression.

Sentence Maker

```
def sentence_maker(phrase):
    # Determine if the sentence is asking a question
    interrogatives = ("how", "what", "why")
    # Return string with first character capitalized
    capitalized = phrase.capitalize()
    if phrase.startswith(interrogatives):
        return f"{capitalized}?"
    else:
        return f"{capitalized}."

print(sentence_maker("how are you today"))
print(sentence_maker("I
like ice cream"))
```

Example run:

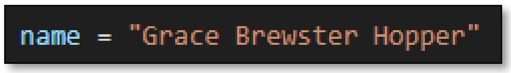


```
How are you today?
I like ice cream.
```

Tutorial 8.4: Initials

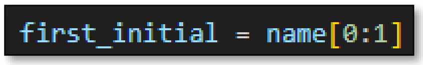
Create a program that asks the user for their first name, middle name, and last name. Print: “Your initials are _ _ _”

Let’s start with a hard coded version.



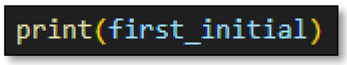
```
name = "Grace Brewster Hopper"
```

Initialize a variable called **name** with the full name "Grace Brewster Hopper." This is the input from which we want to extract initials.



```
first_initial = name[0:1]
```

Extract the first character of the `name` string, which is the first initial. We use Python's string indexing to do this. The **name[0:1]** notation means we start at index 0 (the first character) and go up to, but not including, index 1. This effectively extracts the first character, which is the first initial.



```
print(first_initial)
```

```python
index = name.index(" ")
print(index)
```

Find the index of the first space character in the **name** string. This index represents the position of the first space in the full name.

```python
middle_initial = name[index + 1: index + 2]
print(middle_initial)
```

With the index of the first space found in the previous step, we extract the character immediately after it, which is the middle initial. We use **index + 1** to start at the character following the space and **index + 2** to extract just one character.

```python
index = name.rindex(" ")
print(index)
```

This step is similar to the last step, but now we use **rindex()** to find the index of the last space in the **name** string.

```python
last_initial = name[index + 1: index + 2]
```

With the index of the last space found in the previous step, we extract the character immediately after it, which is the last initial. We use **index + 1** to start at the character following the space and **index + 2** to extract just one character.

```python
initials = first_initial + middle_initial + last_initial
```

Concatenate the initials together.

```python
print(initials)
```

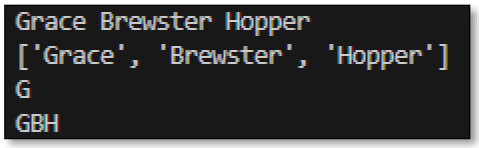
Example run:

```
Grace Brewster Hopper
G
5
B
14
GBH
```

Assignment 1: Use Split for Initials

Use the split method to produce the same results.

Example run:



```
Grace Brewster Hopper
['Grace', 'Brewster', 'Hopper']
G
GBH
```

Assignment 2: Product Code

Create a program for a company that has a product with this item number: "037-00901-0027". 037 is the country code, 00901 is the product code, 00027 is the batch number.

Use the split() method as in the previous assignment.

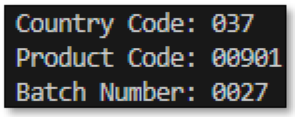
Print:

Country code: _ _ _

Product code: _ _ _ _ _

Batch number: _ _ _ _ _

Example run:



```
Country Code: 037
Product Code: 00901
Batch Number: 0027
```

Lists and Strings

A string is a sequence of characters. A list is a sequence of values. A list of characters is not the same as a string.

list()

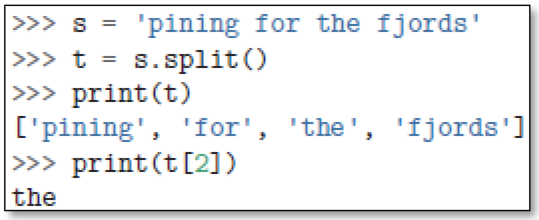
To convert from a string to a list of characters, you can use **list**:

```
s = "spam"
t = list(s)
print(t)
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. Avoid the letter “l” because it looks too much like the number “1”. You might want to use “t”.

split()

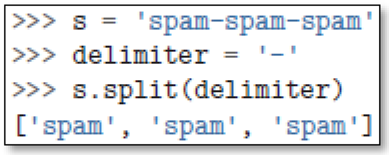
The **list** function breaks a string into individual letters. If you want to break a string into words, use the **split** method:



```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Once you have used split to break the string into a list of words, you can use the index operator (square bracket) to look at a word in the list. By default, split uses the space character to split a string into words.

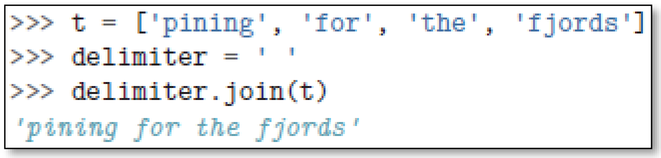
You can call split with an optional argument called a delimiter that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:



```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join()

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:



```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, “”, as a delimiter.

The split method returns a list of the words of a string. The method assumes that words are separated by whitespace, which can be either spaces, tabs or newline characters. Here is an example:

```
s = "Hi! This is a test. "
print(s.split())
['Hi!', 'This', 'is', 'a', 'test.']
```

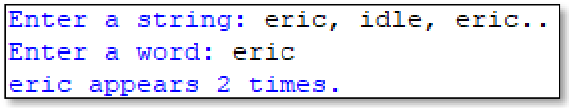
Since **split** breaks up the string at spaces, the punctuation will be part of the words. There is a module called `string` that contains, among other things, a string variable called `punctuation` that contains common punctuation. We can remove the punctuation from a string `s` with the following code:

```
from string import punctuation
s = "cha,ra.c;ter"
for c in punctuation:
   s = s.replace(c, '')
print(s)
character
```

Here is a program that counts how many times a certain word occurs in a string.

```
from string import punctuation
s = input("Enter a string: ")
L = []
for c in punctuation:
   s = s.replace(c, "")
   s = s.lower()
   L = s.split()
word = input("Enter a word: ")
print(f"{word} appears {L.count(word)} times.")
```

Example run:



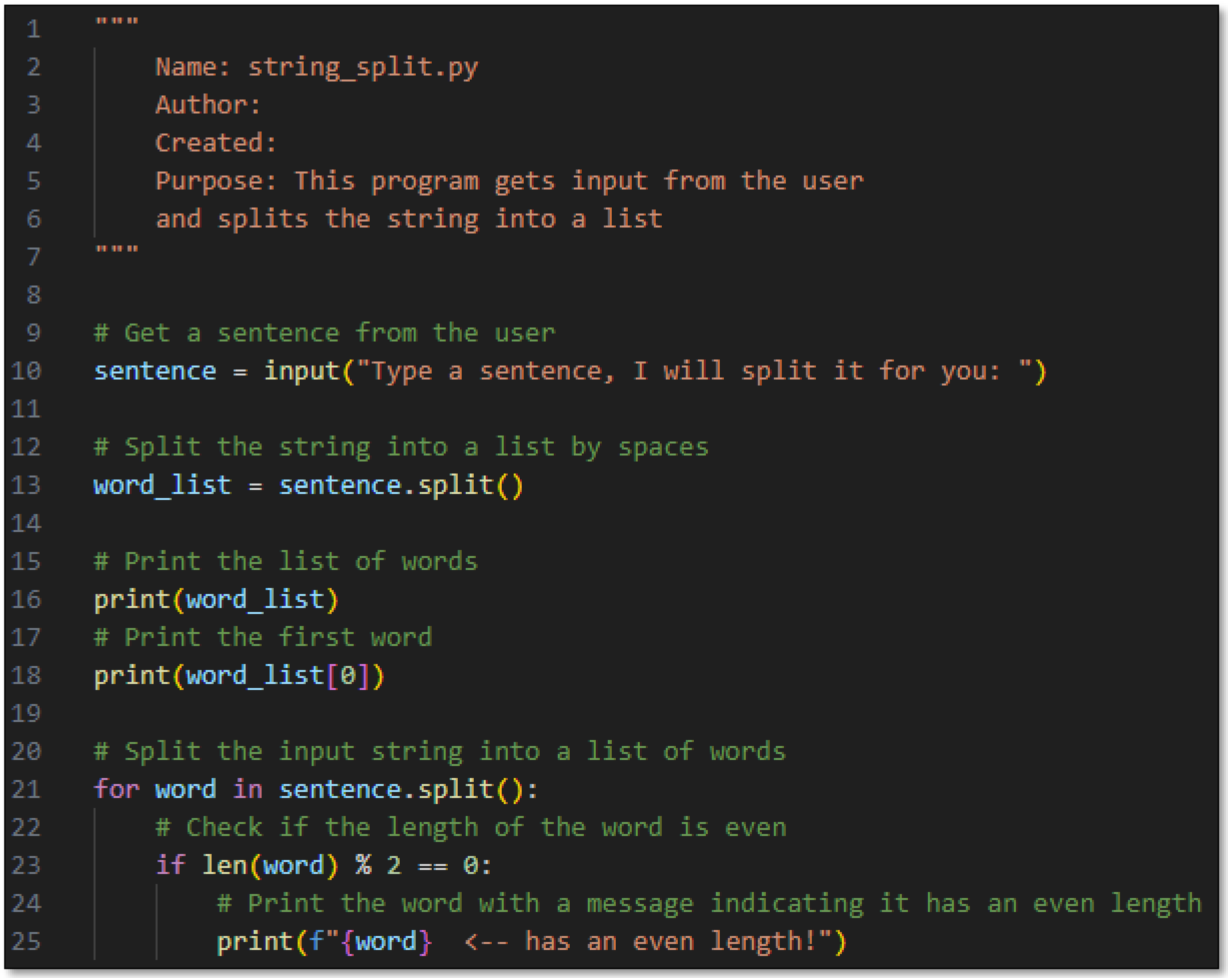
```
Enter a string: eric, idle, eric..
Enter a word: eric
eric appears 2 times.
```

Optional Argument: The **split** method takes an optional argument that allows it to break the string at places other than spaces. Here is an example:

```
s = "1-800-271-8281"
print(s.split("-"))
['1', '800', '271', '8281']
```

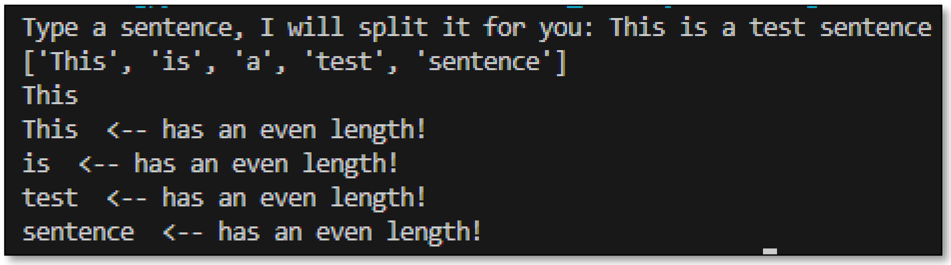
Tutorial 8.4: String Split

The following program **string_split.py** splits a string into a list.



```python
"""
    Name: string_split.py
    Author:
    Created:
    Purpose: This program gets input from the user
    and splits the string into a list
"""

# Get a sentence from the user
sentence = input("Type a sentence, I will split it for you: ")

# Split the string into a list by spaces
word_list = sentence.split()

# Print the list of words
print(word_list)
# Print the first word
print(word_list[0])

# Split the input string into a list of words
for word in sentence.split():
    # Check if the length of the word is even
    if len(word) % 2 == 0:
        # Print the word with a message indicating it has an even length
        print(f"{word}  <-- has an even length!")
```

Example run:



```
Type a sentence, I will split it for you: This is a test sentence
['This', 'is', 'a', 'test', 'sentence']
This
This  <-- has an even length!
is  <-- has an even length!
test  <-- has an even length!
sentence  <-- has an even length!
```

Parsing Strings

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

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

We want 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 find the position of the at-sign in the string. We will find the position of the first space after the at-sign. 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 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
```

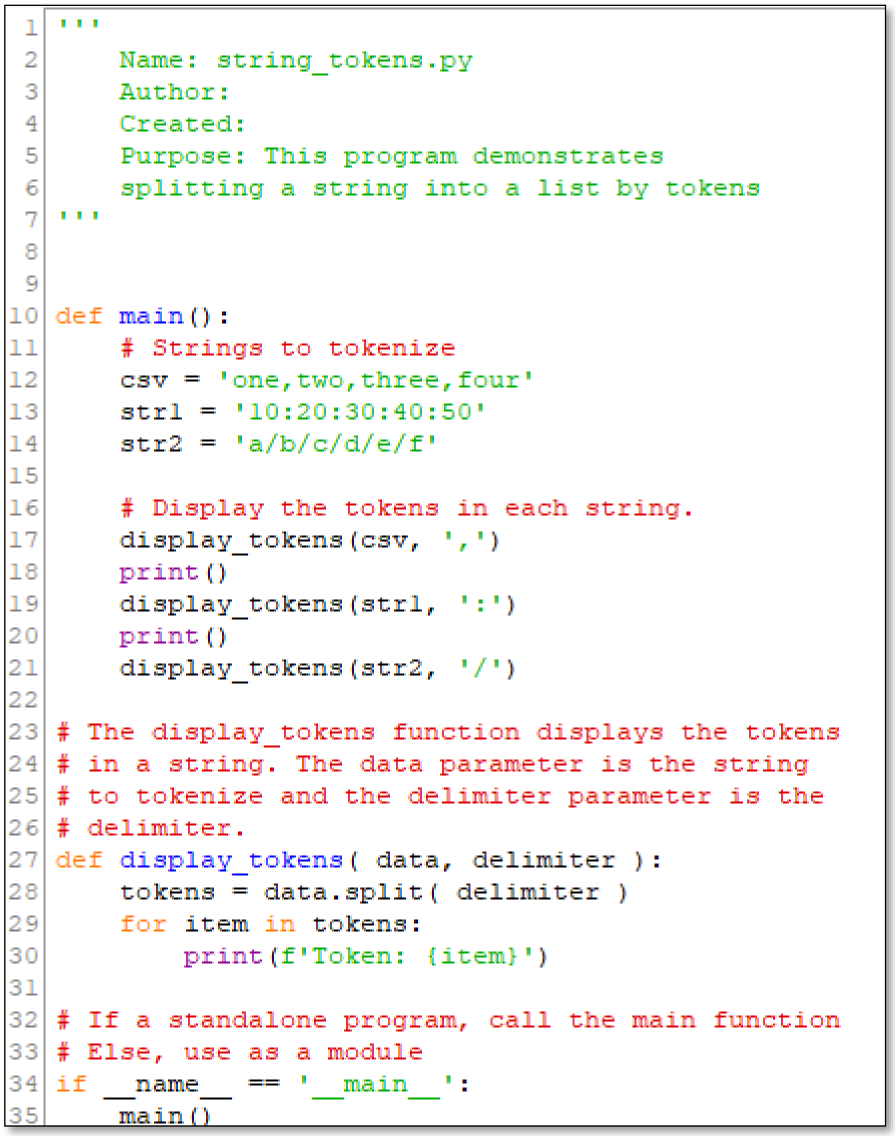
We use a version of the find method which allows us to specify a position in the string where we want `find` to start looking. When we slice, we extract the characters from "one beyond the at-sign through up to but not including the space character".

Tutorial 8.5: String Tokens

At times we will get a text file with delimiters and values (tokens). One way to break a string up into a data structure is to use split().

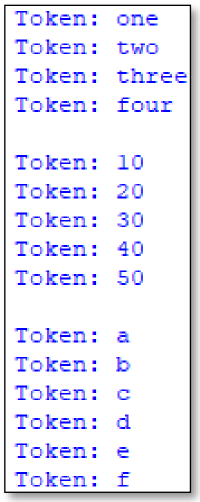
The most common delimiter is a comma ',' from csv (Common Separated Values) files. This is a common data exchange format that can be created in Excel.

Create and run the following program.



```python
'''
    Name: string_tokens.py
    Author:
    Created:
    Purpose: This program demonstrates
    splitting a string into a list by tokens
'''


def main():
    # Strings to tokenize
    csv = 'one,two,three,four'
    str1 = '10:20:30:40:50'
    str2 = 'a/b/c/d/e/f'

    # Display the tokens in each string.
    display_tokens(csv, ',')
    print()
    display_tokens(str1, ':')
    print()
    display_tokens(str2, '/')

# The display_tokens function displays the tokens
# in a string. The data parameter is the string
# to tokenize and the delimiter parameter is the
# delimiter.
def display_tokens( data, delimiter ):
    tokens = data.split( delimiter )
    for item in tokens:
        print(f'Token: {item}')

# If a standalone program, call the main function
# Else, use as a module
if __name__ == '__main__':
    main()
```

Example run:



```
Token: one
Token: two
Token: three
Token: four

Token: 10
Token: 20
Token: 30
Token: 40
Token: 50

Token: a
Token: b
Token: c
Token: d
Token: e
Token: f
```

Escape Characters

The backslash, \, is used to get certain special characters, called escape characters, into your string. There are a variety of escape characters, and here are the most useful ones:

\n the newline character. It is used to advance to the next line. Here is an example:

```
print('Hi\nThere!')
Hi
There!
```

\' for inserting apostrophes into strings. Say you have the following string:

```
s = 'I can't go'
```

This will produce an error because the apostrophe will end the string. You can use \' to get around this:

```
s = 'I can\'t go'
```

Another option is to use double quotes for the string:

```
"s = I can't go"
```

\" analogous to \'

\\ This is used to get the backslash itself. For example:

```
filename = 'c:\\programs\\file.py'
```

\t the tab character

Example 1

An easy way to print a blank line is print(). However, if we want to print ten blank lines, a quick way to do that is the following:

```
print('\n'*9)
```

Note that we get one of the ten lines from the print function itself.

Example 2

Write a program that asks the user for a string and prints out the location of each 'a' in the string.

```
s = input('Enter  some  text:  ')
for i in range(len(s)):
    if s[i]=='a':
        print(i)
```

We use a loop to scan through the string one character at a time. The loop variable i keeps track of our location in the string, and s[i] gives the character at that location. Thus, the third line checks each character to see if it is an 'a', and if so, it will print out i, the location of that 'a'.

Example 3

Write a program that asks the user for a string and creates a new string that doubles each character of the original string. For instance, if the user enters `Hello`, the output should be `HHeellllloo`.

```
s = input('Enter  some  text: ')
doubled_s =  ''
for c in s:
    doubled_s = doubled_s +  c * 2
print(doubled_s)
```

The variable c will run through the characters of s. We use the repetition operator, *, to double each character. We build up the string s in the way.

Example 4

Write a program that asks a user for their name and prints it in the following funny pattern:

```
E El Elv Elvi Elvis
```

We will require a loop because we repeatedly print sections of the string and to print the sections of the string, we will use a slice:

```
name = input('Enter  your name: ')
for i in range(len(name)):
    print(name[:i+1], end='   ')
```

The one trick is to use the loop variable i in the slice. Since the number of characters we need to print is changing, we need a variable amount in the slice.

We want to print one character of the name the first time through the loop, two characters the second time, etc. The loop variable, i, starts at 0 the first time through the loop, then increases to 1 the second time through the loop, etc. Thus we use name[:i+1] to print the first i+1 characters of the name. Finally, to get all the slices to print on the same line, we use the print function's optional argument end=''.

Example 5

Write a program that removes all capitalization and common punctuation from a string s.

```
s = 'This is. A! , lot.'
s = s.lower()
for c  in ',.;:-?!()\'"':
    s = s.replace(c, '')
print(s)
```

The way this works is for every character in the string of punctuation, we replace every occurrence of it in s with the empty string, ''. One technical note here: We need the ' character in a string. As described in the previous section, we get it into the string by using the escape character \'.

Example 6

Write a program that, given a string that contains a decimal number, prints out the decimal part of the number. For instance, if given 3.14159, the program should print out .14159.

```
s = input('Enter your decimal number:  ')
print(s[s.index('.')+1:])
```

The key here is the index method will find where the decimal point is. The decimal part of the number starts immediately after that and runs to the end of the string, so we use a slice that starts at s.index('.')+1.

Here is another, more mathematical way, to do this:

```
from math import floor
num = eval(input('Enter your decimal number: ')
print(num - floor(num))
```

One difference between the two methods is the first produces a string, whereas the second produces a number.

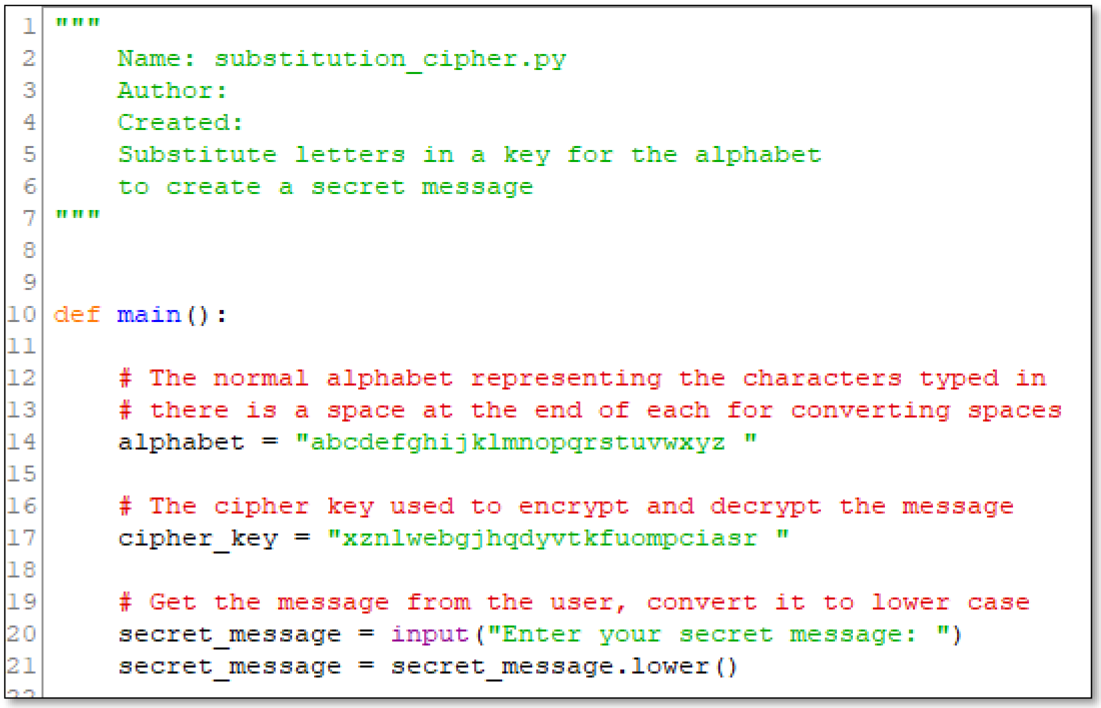
8.6: Substitution Cipher

A simple and very old method of sending secret messages is the substitution cipher. Each letter of the alphabet gets replaced by another letter of the alphabet, say every **a** gets replaced with an **x**, and every **b** gets replaced by a **z**, etc.

This tutorial shows one way to implement a substitution cipher named **substitution_cipher.py**

You can copy and paste the following code to save some time.

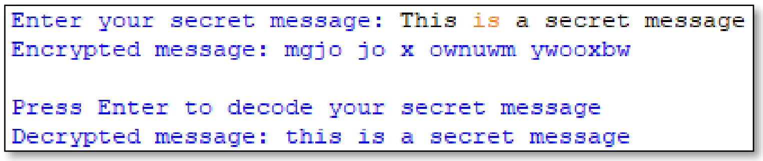
```
# The normal alphabet representing the characters typed in
# A space is included at the end of each string
alphabet = "abcdefghijklmnopqrstuvwxyz "

# The cipher key used to scramble the message
cipher_key = "xznlwebgjhqdyvtkfuompciasr "
```



```
"""
    Name: substitution_cipher.py
    Author:
    Created:
    Substitute letters in a key for the alphabet
    to create a secret message
"""


def main():

    # The normal alphabet representing the characters typed in
    # there is a space at the end of each for converting spaces
    alphabet = "abcdefghijklmnopqrstuvwxyz "

    # The cipher key used to encrypt and decrypt the message
    cipher_key = "xznlwebgjhqdyvtkfuompciasr "

    # Get the message from the user, convert it to lower case
    secret_message = input("Enter your secret message: ")
    secret_message = secret_message.lower()
```

```python
#---------------------- ENCRYPT MESSAGE ------------------------------#
    encrypted_message = ""
    # Go through the string one character at a time
    for character in secret_message:

        if character.isalpha() or character.isspace():
            # If the character is a letter or a space
            # Find the index number of the character
            index = alphabet.index(character)

            # Use the index number to find the corresponding
            # cipher key character and append it to the
            # encrypted_message string , scrambling the message
            encrypted_message = encrypted_message + cipher_key[index]
        else:
            # If the character is not a letter, leave it alone
            # Add it to the string as is
            encrypted_message = encrypted_message + cipher_key[index]

    print(f"Encrypted message: {encrypted_message}")
    # Time to decode our secret message
    input("\nPress Enter to decode your secret message ")

#---------------------- DECRYPT MESSAGE ------------------------------#
    original_message = ""
    # Go through the string one character at a time
    for character in encrypted_message:

        if character.isalpha() or character.isspace():
            # If the character is a letter or a space
            # Find the index number of the character
            index = cipher_key.index(character)

            # Use the index number to get the corresponding
            # alphabet key character, unscrambling the message
            original_message = original_message + alphabet[index]
        else:
            # If the character is not a letter, leave it alone
            original_message = original_message + alphabet[index]
    print(f"Decrypted message: {original_message}")


# Call the main function
if __name__ == "__main__":
    main()
```

Example run:



```
Enter your secret message: This is a secret message
Encrypted message: mgjo jo x ownuwm ywooxbw

Press Enter to decode your secret message
Decrypted message: this is a secret message
```

The string **cipher_key** is a random reordering of the alphabet.

The tricky part of the program is the for loop. It goes through the message one character at a time, and, for every letter it finds, it replaces it with the corresponding letter from the key. This is accomplished by using the index method to find the position in the alphabet of the current letter and replacing that letter with the letter from the key at that position. All non-letter characters are copied as is. The program uses the isalpha() method to tell whether the current character is a letter or not. It uses the isapace() method to find spaces.

The code to decipher a message is nearly the same. Change the key[alphabet.index(c)] to alphabet[key.index(c)].

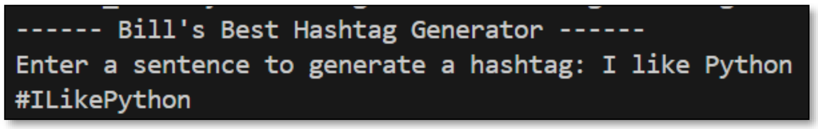
Assignment 1: Choose Your Own

Here are some assignment options. Choose the one that interests you the most.

Hashtag Generator

- Create a Python program called hash_tag_generator.py in OOP.
- Write a method that converts a given sentence into a hashtag.
- The method should capitalize each word, remove spaces, and add a # at the beginning.
- Limit the length to 140 characters.

Example run:



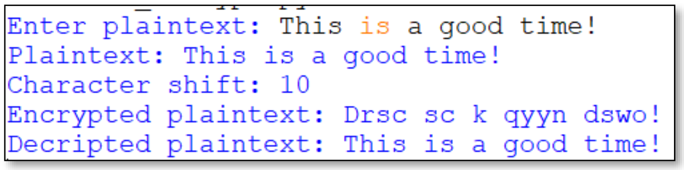
```
------ Bill's Best Hashtag Generator ------
Enter a sentence to generate a hashtag: I like Python
#ILikePython
```

Caesar Cipher

Modify the Substitution Cipher tutorial to implement a Caesar Cipher, which shifts each letter in the string by a specified number of places. Make both encoding and decoding methods.

Example: "abc", shift = 3 → "def"

Example run:

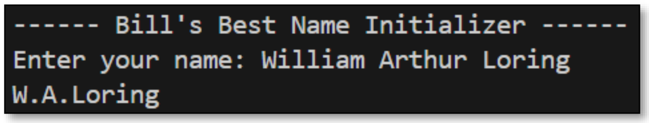


```
Enter plaintext: This is a good time!
Plaintext: This is a good time!
Character shift: 10
Encrypted plaintext: Drsc sc k qyyn dswo!
Decripted plaintext: This is a good time!
```

Name Initializer

- Create a Python program named **name_initializer.py**
- Write a function that takes a full name and returns it in a shortened format with initials.
- Optionally, add middle names if they exist.

Example: "Jane Ann Doe" → "J.A. Doe"

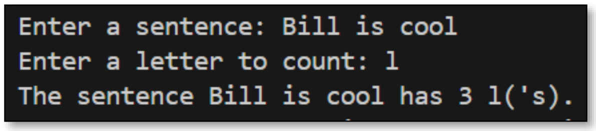


```
------ Bill's Best Name Initializer ------
Enter your name: William Arthur Loring
W.A.Loring
```

Character Count

This program gets input from the user and counts the number of times a letter appears.

Example run:



```
Enter a sentence: Bill is cool
Enter a letter to count: l
The sentence Bill is cool has 3 l('s).
```

Glossary

counter A variable used to count something, usually initialized to zero and then incremented.

empty string A string with no characters and length 0, represented by two quotation marks.

flag A boolean variable used to indicate whether a condition is true or false.

invocation A statement that calls a method.

immutable The property of a sequence whose items cannot be assigned.

index An integer value used to select an item in a sequence, such as a character in a string.

item One of the values in a sequence.

method A function that is associated with an object and called using dot notation.

object Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

search A pattern of traversal that stops when it finds what it is looking for.

sequence An ordered set; that is, a set of values where each value is identified by an integer index.

slice A part of a string specified by a range of indices.

traverse To iterate through the items in a sequence, performing a similar operation on each.

Assignment Submission

1. Attach the pseudocode or create a TODO.
2. Attach all tutorials and assignments.
3. Attach screenshots showing the successful operation of each tutorial program.
4. Submit in Blackboard.