CSC 171 - Introduction to Computer Programming

Lecture #6 - Stringing Along – Using Character and String Data

How Do Computer Handle Character Data?

- Like all other data that a computer handles, characters are stored in numeric form.
- A particular code represents a particular character.
- The most commonly used code was ***ASCII*** (***A***merican ***S***tandard ***C***ode for ***I***nformation ***I***nterchange). Python uses a code called ***Unicode***.

Example: Comparing Characters

```
char1 = 'a'
char2 = 'b'
char3 = "A" # Can be enclosed with ' ' or " "
if char1 > char2 :
    print("Very good") # also true for strings
else :
    print('Very bad')
if char1 > char3 :
    print("Very good")
else :
    print("Very bad")
```

What are Strings?

- A collection of characters that are read and written together to form words, numbers and so on are called ***strings***.
- Strings have certain methods that can be used to manipulate them. At the same time, they can be used in some ways that are like the basic data type in Python, such as **int** and **float**.
- Individual characters in Python are considered string of length 1.

Assigning a Value to String

- Every character is "mapped" (associated) with an integer
- UTF-8, subset of Unicode, is such a mapping
- The function **`ord()`** takes a character and returns its UTF-8 integer value, **`chr()`** takes an integer and returns the UTF-8 character.

Subset of UTF-8

Char	Dec	Char	Dec	Char	Dec
SP	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104

Subset of UTF-8 (continued)

Char	Dec	Char	Dec	Char	Dec
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116

Assigning a Value to String

- A value can be assigned to a string by putting the characters inside single or double quotes:

```
>>> s = "This is the first"
>>> print (s)
This is the first
>>> t = 'This is the second'
>>> print(t)
This is the second
>>>
```

Python String Input/Output - An Example

```
s = input(("Enter your string"))
print("Your string is \"", s, "\".")
>>>
Enter your stringThis is the first
Your string is " This is the first ".
>>>
```

Example: Comparing Strings

```
string1 = "as"
string2 = "As"
string3 = "apple" # Can be enclosed with ' ` or "
"
if string1 > string2 :
    print("Very good") # also true for strings
else :
    print('Very bad')
if string1 > string3 :
    print("Very good")
else :
    print("Very bad")
```

Concatenation and Repetition

- Concatenation is the operation where we join two strings together into one longer string.

```
s = "The " + "Second"
print(s)
```

- will print `"The Second"`
- Repetition is the operation where we create a string that contains the same sequence of characters multiple times.

```
s = "my" * 3
print(s)
```

will print `"mymymy"`

The Python String Functions and Methods

- A particular element of the string is accessed by the index of the element surrounded by square brackets

```
hello_str = 'Hello World'
print(hello_str[1])  ⇒       prints e
print(hello_str[-1]) ⇒       prints d
print(hello_str[11]) ⇒       ERROR
```

Slicing – The Rules

- Slicing is the ability to select a subsequence of the overall sequence
- Uses the syntax **`[start : finish]`**, where:
  - **`start`** is the index of where we start the subsequence
  - **`finish`** is the index of **one after** where we end the subsequence
- If either **`start`** or **`finish`** are not provided, it defaults to the beginning of the sequence for **`start`** and the end of the sequence for **`finish`**

Half Open Range for Slices

- Slicing uses what is called a half-open range
- The first index is included in the sequence
- The last index is one **after** what is included

Indexing and Slicing

```
helloString[6:10]
```

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

first (6) last (10)

Indexing and Slicing

```
helloString[6:]
```

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

first (6) last (past 10)

```
helloString[:5]
```

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

first (0) last (5)

Indexing and Slicing

```
helloString[-1]
```

Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10
	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

Last

Indexing and Slicing

```
helloString[3:-2]
```

Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10

First

Last

Extending Slicing

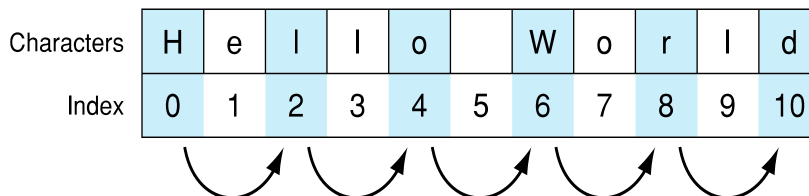
- Also takes three arguments:
  - **`[start:finish:countBy]`**
- Defaults are:
  - **`start`** is beginning, **`finish`** is end, **`countBy`** is 1.
- Example

```
my_str = 'hello world'
my_str[0:11:2] ⇒ 'hlowrd'
```

Every other letter

Extended Slicing

`helloString[::2]`



Some Python Idioms

- Idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk
- How to make a copy of a string:

```
>>> my_str = 'hi mom'
>>> new_str = my_str[:]
>>> print(new_str)
hi mom
>>>
```

Some Python Idioms

- Idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk
- How to reverse a string

```
>>> my_str = "madam I'm adam"
>>> reverseStr = my_str[::-1]
>>> print(reverseStr)
mada m'I madam
>>>
```

Some Python Idioms

- Idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk
- How to make a copy of a string:

```
my_str = 'hi mom'
new_str = my_str[:]
```

- How to reverse a string

```
my_str = "madam I'm adam"
reverseStr = myStr[::-1]
```

Sequences are Iterable

- The **for** loop iterates through each element of a sequence in order. For a string, this means character by character:

```
>>> for char in 'Hi mom' :
    print(char, type(char))

H <class 'str'>
i <class 'str'>
  <class 'str'>
m <class 'str'>
o <class 'str'>
m <class 'str'>
>>>
```

Basic String Operations

```
s = 'spam'
```

- Length operator **`len()`**

  `len(s)` ⇒ `4`

- **`+`** is concatenate

  `new_str = 'spam' + '-' + 'spam-'`

  print(new_str) ⇒ `spam-spam-`

- **`*`** is repeat, the number is how many times

  `new_str * 3`

  `'spam-spam-spam-spam-spam-spam-'`

Membership Operations

- Can check to see if a substring exists in the string, the **`in`** operator. Returns **`True`** or **`False`**

  `my_str = 'aabbccdd'`

  `'a' in my_str` ⇒ `True`

  `'abb' in my_str` ⇒ `True`

  `'x' in my_str` ⇒ `False`

Some Details

- Both `+` and `*` on strings makes a new string, does not modify the arguments
- Order of operation is important for concatenation, irrelevant for repetition
- The types required are specific. For concatenation you need two strings, for repetition a string and an integer

What Does `a + b` Mean?

- What operation does the above represent? It depends on the types!
  - Two strings, concatenation
  - Two integers addition
- The operator `+` is ***overloaded***.
  - The operation `+` performs depends on the types it is working on

Functions

- A function is a program that performs some operation. Its details are hidden (encapsulated), only it's interface provided.
- A function takes some number of inputs (arguments) and returns a value based on the arguments and the function's operation.

The **`type`** Function

- You can check the type of the value associated with a variable using **`type`**
  - **`my_str = 'hello world'`**
  - **`type(my_str)`** ⇒ **`<type 'str'>`**
  - **`my_str = 245`**
  - **`type(my_str)`** ⇒ **`<type 'int'>`**

String Function : `len`

- The **`len`** function takes as an argument a string and returns an integer, the length of a string.

```
my_str = 'Hello World'
len(my_str)   11  # space
counts!!
```

String Method

- A **method** is a variation on a function
  - like a function, it represents a program
  - like a function, it has input arguments and an output
- Unlike a function, it is applied in the context of a particular object.
- This is indicated by the dot notation invocation

Python String Methods

- **`s.strip()`** - Returns s with leading and trailing white space characters removed.
- **`s.lstrip()`** – Returns s with leading white space characters removed.
- **`s.rstrip()`** – Returns s with trailing white space characters removed.
- **`s.find(t)`** - Returns the starting position of the first occurrence of the substring **`t`** within **`s`**.
- **`s.rfind(t)`** – Returns the startingposition of the rightmost occurrence of the substring **`t`** within **`s`**.

`s.strip()`

- Returns s with leading and trailing white space
- characters removed.

```
s = " This Is The First "
s = s.strip();
print("My String is \'", s, "\'")
print ("It has ", len(s), " characters.")
```

- The output is:

```
My String is 'This Is The First'
It has 17 characters.
```

s.lstrip()

- Returns s with leading white space characters removed.

```
s = " This Is The First "
s = s.lstrip();
print("My String is \'", s, "\'")
print ("It has ", len(s), " characters.")
```

- The output is:

```
My String is ' This Is The First  '
It has  18  characters.
```

s.rstrip()

- Returns s with trailing white space characters removed.

```
s = " This Is The First "
s = s.rstrip();
print("My String is \'", s, "\'")
print ("It has ", len(s), " characters.")
```

- The output is:

```
My String is '  This Is The First '
It has  18  characters.
```

s.find(t)

- **s.find()** can be used to find where a substring appears within **s**.
- Example

```
s = "John Francis Xavier Smith"
i = s.find("Fran");
t = s[i:i+7]
print(t, "\'begins at position", i)
```

- Output

```
Francis 'begins at position 5
```

s.rfind(t)

- **s.rfind()** can be used to find where the last occurrence of a substring appears within **s**.
- Example

```
s = "John Francis Xavier Smith"
i = s.find(“Xav");
t = s[i:i+7]
print(t, "\'begins at position", i)
```

- Output

```
Francis 'begins at position 20
```

Dot Notation

- In generation, dot notation looks like:
  `object.method(...)`
- It means that the object in front of the dot is calling a method that is associated with that object's type.
- The method's that can be called are tied to the type of the object calling it. Each type has different methods.

Python String Methods

- **`s.replace(t, u)`** - Returns **`s`** with the next occurrence of **`t`** replaced by **`c`**’s text and trailing white space characters removed.
- **`s.ljust(x)`** – Returns **`s`** left justified within a string of length **`x`**.
- **`s.rjust(x)`** – Returns **`s`** right justfied within a string of length **`x`**.
- **`s.center(x)`** - Returns **`s`** with its text centered within a string of length **`x`**.

s.replace(t, u)

- **s.replace** will return a string with every occurrence of specified its text replaced.
- You can include the number of times you wish this replacement to occur
- Example

```
s = "James Roosevelt Roosevelt"
t = s.replace("Roosevelt", "Rosie")
print("\"", t, "\"")

t = s.replace("Roosevelt", "Rosie", 1)
print("\"", t, "\"")
```

s.ljust(x)

- **s.ljust** will return a string with the text left adjusted within a field of a given width
- Example

```
s = "John Francis Xavier Smith"
i = len(s)
print(i)
s = s.ljust(i+5);
print("\"", s, "\"")
```

- Output

```
25
" John Francis Xavier Smith       "
```

s.rjust(x)

- **s.ljust** will return a string with the text left adjusted within a field of a given width
- Example

```
s = "John Francis Xavier Smith"
i = len(s)
print(i)
s = s.rjust(i+5);

print("\"", s, "\"")
```

- Output

```
25
"      John Francis Xavier Smith "
```

s.center(x)

- **s.center** will return a string with the text centered within a field of a given width
- Example

```
s = "John Francis Xavier Smith"
i = len(s)
print(i)
s = s.center(i+5);

print("\"", s, "\"")
```

Output

```
25
"   John Francis Xavier Smith    "
```

Chaining Methods

Methods can be chained together.

- Perform first operation, yielding an object
- Use the yielded object for the next method

```
my_str = 'Python Rules!'
my_str.upper() ⇒ 'PYTHON RULES!'
my_str.upper().find('O')
⇒ 4
```

Nesting Methods

- You can “nest” methods, that is the result of one method as an argument to another
- Remember that parenthetical expressions are did “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument
- `a_str.find('t', a_str.find('t')+1)`
- Translation: find the second `'t'`.

עד כאן

Python String Methods

- **`s.count(t)`** - Returns the number of occurrences of the substring t inside **`s`**.
- **`s.startswith(t)`** – Returns True if **`s`** begins with the substring t.
- **`s.endswith(t)`** – Returns True if **`s`** ends with the substring t.

s.count(t)

- **`s.count(t)`** returns the number of occurences of the string **`t`** in **`s`** that do not overlap.
- Example

```
>>> s = "Aiiieie!"
>>> print(s.count("ii"))
1
>>>
```

s.startswith(t)

- Returns True if **s** begins with the substring t.

```
>>> s = \
 "the rain in Spain stays mainly in the plain"
>>> print(s.startswith("the"))
True
>>>
```

- Using the optional second argument:

```
>>> print(s.startswith("in", 6))
True
>>>
```

s.endswith(t)

- Returns True if **s** begins with the substring t.
- Example

```
>>> s = \
  "The rain in Spain stays mainly in the plain"
>>> print(s.endswith("in"))
True
>>> print (s.endswith("in", 40))
True
>>> print(s[0:41])
The rain in Spain stays mainly in the pla
>>>
```

`s.endswith(t)`

- If you add two optional argument they are the beginning and end of the substring to be checked
- Example

```
>>> s="programming is easy to learn."
>>> print(s.endswith('learn.', 7))
True
>>> print(len(s))
29
>>> print(s.endswith('learn.', 7, 26))
False
>>> print(s[7:27])
ming is easy to lear
>>> print(s.endswith('learn.', 7, 27))
False
```

Python String Methods

- **`s.isalpha()`** - Returns True if all the characters in **`s`** are letters.
- **`s.isdigit()`** - Returns True if all the characters in **`s`** are digits.
- **`s.isupper()`** - Returns True if all the letters in **`s`** are capital letters.
- **`s.islower()`** - Returns True if all the letters in **`s`** are lower-case letters.
- **`s.istitle()`** – Returns True if all the words in **`s`** begin with capital letters and the rest are lower case.

s.isalpha()

- Returns True if all the characters in **s** are letters.

```
>>> s="The quick brown fox"
>>> print(s.isupper())
False
>>> print(s[0].isupper())
True
>>> s = "HELP"
>>> print(s.isupper())
True
>>> s ="ALL IS LOST"
>>> print(s.isupper())
True
>>>
```

s.digit()

- Returns True if all the characters in **s** are digits.
- Examples

```
>>> x = "13"
>>> print(x.isdigit())
True
>>> x = "9/13"
>>> print(x.isdigit())
False
>>>
```

s.isupper()

- Returns True if all the letters in **s** are capital letters.
- Example

```
>>> s = "LETS GO METS"
>>> print(s.isupper())
True
>>> s = "LET'S GO, METS"
>>> print(s.isupper())
True
>>> print(s)
```

s.islower()

- Returns True if all the letters in **s** are lower-case letters.
- Examples

```
>>> s = "be very quiet - the baby is sleeping"
>>> print(s.islower())
True
>>> s = "be very quiet - Junior is sleeping"
>>> print(s.islower())
False
>>> print(s[0:16].islower())
True
>>>
```

`s.istitle()`

- Returns True if all the words in **`s`** begin with capital letters and the rest are lower case.
- Examples

```
>>> s = "A Tale of Two Cities"
>>> print(s.istitle())
False
>>> s = "A Tale Of Two Cities"
>>> print(s.istitle())
True
>>>
```

Strings Are Immutable

- strings are immutable, that is you cannot change one once you make it:

```
a_str = 'spam'
a_str[1] = '1' ⇒ ERROR
```

- However, you can use it to make another string (copy it, slice it, etc.)

```
new_str = a_str[:1] + '1' + a_str[2:]
a_str   'spam'
new_str = 'slam'
```

Python String Methods

- **`s.upper()`** - Returns a string with s's contents in upper case.
- **`s.lower()`** - Returns a string with s's contents in lower case.
- **`s.swapcase()`** - Returns a string with s's contents in converted from upper to lower case, and from lower to upper case.

s.upper()

- Returns a string with s's contents in upper case.
- Example

```
>>> print(s)
a tale of two cities
>>> print(s.upper())
A TALE OF TWO CITIES
>>>
```

s.lower()

- Returns a string with s's contents in lower case.
- Example

```
>>> print(s)
A Tale Of Two Cities
>>> print(s.lower())
a tale of two cities
>>>
```

s.swapcase()

- Returns a string with s's contents in converted from upper to lower case, and from lower to upper case.
- Example

```
>>> print(s)
A Tale Of Two Cities
>>> print(s.swapcase())
a tALE oF tWO cITIES
>>>
```

Python String Methods

- **`s.capitalize()`** - Returns a string with s's letters in lower case, except for the first character if it is a letter.
- **`s.title()`** – Returns a string with the first letter in s's words in upper case, the rest in lower case.

`s.capitalize()`

- Returns a string with s's letters in lower case, except for the first character if it is a letter.

```
>>> print(s)
A Tale Of Two Cities
>>> print(s.capitalize())
A tale of two cities
>>> s = "-" + s
>>> print(s)
-A Tale Of Two Cities
>>> print(s.capitalize())
-a tale of two cities
```

`s.title()`

- Returns a string with the first letter in s's words in upper case, the rest in lower case.

```
>>> s = "a tale of TWO cities"
>>> print(s.title())
A Tale Of Two Cities
>>> s = "-a tale of two cities"
>>> print(s.title())
-A Tale Of Two Cities
>>>
```

Example – Detecting Palindromes

- A palindrome is a word or phrase that is the same forward and backward
  - Input: The phrase being tested
  - Output: Message telling the user if it is or is not a palindrome
- Algorithm

1. Read a phrase
2. Tell the user if it is or isn't a palindrome

Refining the Algorithm

1. Read a phrase
2. Tell the user if it is or isn't a palindrome

2.1 Strip out any non-letters
2.2 Convert it to lower case
2.3 Print whether it's a palindrome

Refining the Algorithm

1. Read a phrase

2.1 Strip out any non-letters
2.2 Convert it to lower case
2.3 Print whether it's a palindrome

2.1 For every character in the phrase:
2.1.1 If it is a letter
2.1.2 Copy it over into the string

1. Read a phrase

2.1 For every character in the phrase:

2.1.1 If it is a letter

2.1.2 Copy it over into the string

2.2 Convert it to lower case

2.3 Print whether it's a palindrome

2.3 If the forward and reverse strings are equal
2.3.1 Print that it is a palindrome
2.3.2 Else print that it isn't a palindrome

1. Read a phrase

2.1 For every character in the phrase:

2.1.1 If it is a letter

2.1.2 Copy it over into the string

2.2 Convert it to lower case

2.3 If the forward and reverse strings are equal

2.3.1 Print that it is a palindrome

2.3.2 Else print that it isn't a palindrome

```
print("Type in a phrase.  We'll "\
              + " tell you if it's a palindrome")
phrase = input("What's your phrase?")
```

```
print("Type in a phrase.  We'll "\
              + " tell you if it's a palindrome")
phrase = input("What's your phrase?")
```

2.1 For every character in the phrase:

2.1.1 If it is a letter

2.1.2 Copy it over into the string

2.2 Convert it to lower case

2.3 If the forward and reverse strings are equal

2.3.1 Print that it is a palindrome

2.3.2 Else print that it isn't a palindrome

```
newphrase = ""
for letter in phrase :
    if letter.isalpha():
        newphrase = newphrase + letter
```

```
print("Type in a phrase.  We'll "\
              + " tell you if it's a palindrome")
phrase = input("What's your phrase?")
newphrase = ""
for letter in phrase :
    if letter.isalpha():
        newphrase = newphrase + letter
```

2.2 Convert it to lower case

2.3 If the forward and reverse strings are equal

2.3.1 Print that it is a palindrome

2.3.2 Else print that it isn't a palindrome

```
newphrase = newphrase.lower()
```

```
print("Type in a phrase.  We'll "\
              + " tell you if it's a palindrome")
phrase = input("What's your phrase?")
newphrase = ""
for letter in phrase :
    if letter.isalpha():
        newphrase = newphrase + letter
newphrase = newphrase.lower()
```

2.3 If the forward and reverse strings are equal

2.3.1 Print that it is a palindrome

2.3.2 Else print that it isn't a palindrome

```
if newphrase == newphrase[::-1] :
```



```
print("Type in a phrase.  We'll "\
              + " tell you if it's a palindrome")
phrase = input("What's your phrase?")
newphrase = ""
for letter in phrase :
    if letter.isalpha():
        newphrase = newphrase + letter
newphrase = newphrase.lower()
if newphrase == newphrase[::-1] :
```

2.3.1 Print that it is a palindrome

2.3.2 Else print that it isn't a palindrome


```
print("Your phrase, \"", phrase, "\" is a palindrome")
else :
    print("Your phrase, \"", phrase, "\", is a palindrome")
```

Palindrome.py

```
#  Detect a Palindrome

#  Print instructions and read in a phrase
print("Type in a phrase.  We'll tell you if it's
a palindrome")
phrase = input("What's your phrase?")

#  We copy over the phrase one letter at a time
#  We need to remove anything that is not a
letter
newphrase = ""
for letter in phrase :
    if letter.isalpha():
        newphrase = newphrase + letter
```

```
# Convert it to all lower case to make it case
insensitive
newphrase = newphrase.lower()

#  Print a message telling the user if it is or
is not a palindrome.
if newphrase == newphrase[::-1] :
    print("Your phrase, \"", phrase, \
              "\" is a palindrome")
else :
    print("Your phrase, \"", phrase, \
              "\", is a palindrome")
```

Example: Writing Changing a Form Letter

- Let's write a program to read a file and change every occurrence of the name "John" to "Robert"
- Initial algorithm:
  1. Instruct the user
  2. Change every occurrence on each line of "John" to "Robert

Refining the Form Letter Algorithm

1. Instruct the user
2. Change every occurrence on each line of "John" to "Robert

2.1 Get the first line
2.2 As long as it isn't "The End", replace every occurrence of John with Robert

Refining the Form Letter Algorithm

1. Instruct the user
2.1 Get the first line
2.2 As long as it isn't "The End", replace every occurrence of John with Robert

2.2 WHILE the line ≠ "The End"
2.2.1 Replace each occurrence of "John" with Robert
2.2.2 Print the new line
2.2.3 Get the next line

Refining the Form Letter Algorithm

1. Instruct the user
2.1 Get the first line
2.2 WHILE the line ≠ "The End"
2.2.1 Replace each occurrence of "John" with Robert
2.2.2 Print the new line
2.2.3 Get the next line

```
outString  = inString.replace("John", "Robert")
```

Refining the Form Letter Algorithm

1. Instruct the user
2.1 Get the first line
2.2 WHILE the line ≠ “The End”

```
outString  = inString.replace("John", "Robert")
```

2.2.2 Print the new line
2.2.3 Get the next line

```
inString = input("Please begin typing. " \
            + "End by typing \'The End\'\n")
```

Refining the Form Letter Algorithm

```
inString = input("Please begin typing. " \
            + "End by typing \'The End\'\n")
```

2.2 WHILE the line ≠ “The End”

```
outString  = inString.replace("John", "Robert")
```

2.2.2 Print the new line
2.2.3 Get the next line

```
while inString != "The End" :
```

Refining the Form Letter Algorithm

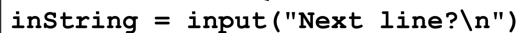
```
inString = input("Please begin typing. " \
            + "End by typing \'The End\'\n")
while inString != "The End" :
outString  = inString.replace("John", "Robert")
```

2.2.2 Print the new line
2.2.3 Get the next line

Refining the Form Letter Algorithm

```
inString = input("Please begin typing. " \
            + "End by typing \'The End\'\n")
while inString != "The End" :
outString  = inString.replace("John", "Robert")
print(outString)
```

2.2.3 Get the next line



Example: **`ChangeLetter.py`**

```
# Change every occurrence of "John" in the
# text of a form letter to "Robert"

# Prompt the user and instruct him.her how
# to indicate the end of the letter
inString = input("Please begin typing. " \
            + "End by typing \'The End\'\n")
# Keep changing as long as (s)he didn't
# type "the end"
while inString != "The End" :
    outString = inString.replace("John", \
            "Robert")
    print(outString)
    inString = input("Next line?\n")
```