Chapter XIV

String Processing

Chapter XIV Topics

14.1 Introduction

Strings are a set of characters of every conceivable arrangement and size. Strings are everywhere, both inside and outside the computer world. A sentence is a set of characters. A page is a set of sentences. A book is a set of pages. A library is a set of books. Given enough computer memory, an entire library can be stored in a computer. Word processing term papers, sending email messages, responding to online surveys, and even writing Python programs in an IDE like **jGRASP** all involve *string processing*. Every software package on the market includes string-processing components. Every programming language has special features that facilitate the manipulation of strings, and Python is no different. Finally, let us not forget that every program you write is one big collection of strings that work together, hopefully, to generate some desired output.

You have been working with strings for quite some time now. You may be under the impression that "string" or **str** is a *simple data type*. This is not surprising. Consider the variable definitions in Figure 14.1.

Figure 14.1

```
age = 50
gpa = 3.785
finished = True
title = "Exposure Computer Science"
```

On the first line, **age** is storing a single integer value; on the second line, **gpa** is storing a single real number value; on the third line, **finished** is storing a single Boolean value; and on the fourth line, **title** is storing a single string value. All 4 lines seem to be perfect examples of *simple data types*.

Well, the first 3 lines are definitely examples of *simple data type*. No question. However, with the last line there can be some argument. Some people would say it is an example of simple data type because **title** stores one string value. Fair enough, but other people argue the last line is an example of a *data structure* because **title** stores several character values. This is also true.

So which one is correct? Well, here is my opinion. I would say that string is a special *data structure* that is designed in such a way that is can also be used as a *simple data type*. We have been using string as a simple data type for more than a semester. In this chapter, we are going to look at the capabilities of the string data structure and how they make *String Processing* possible.

Before we get deep into *String Processing*, we need to review some string related terms from the first semester. Look closely at the first line of the definition of a *string* and see if it makes more sense now.

String Definition

A *string* is a set of characters that behaves as a single unit.

The characters in a string can include upper-case letters, lower-case letters, numerical digits and a large set of symbols for a variety of purposes like:

```
!   @   #   $   %  ^   &   *   (   )   _   +
```

String Variables vs. String Literals

A *string literal* is a set of characters delimited with quotations.

```
name = "John Smith"
```

name is the string variable.

"John Smith" is the string literal.

14.2 String Operators

Review of String Concatenation

We are going to look at several string operators. We will start by reviewing the 2 that are used for *string concatenation*. Computer science uses the fancy word *concatenation* to the joining together of two or more strings. Figure 14.2 compares the difference between mathematical addition and string concatenation.

Figure 14.2

Mathematical Addition	String Concatenation
`x = 100 + 200`	`x = "100" + "200"`

`x = 100`
`x += 200`
	`x = "100"`
	`x += "200"`

In both cases **x** now stores **300**.	In both cases **x** now stores **"100200"**.

Program **StringOperators01.py**, shown in Figure 14.3, reviews *string concatenation* with the *overloaded* plus (**+**) operator.

Figure 14.3

```
# StringOperators01.py
# This program reviews "Concatenation"
# with the overloaded <+> operator.


s1 = "Argentine"
s2 = "Tango"
s3 = s1 + "  " + s2
print()
print(s3)

s4 = "100"
s5 = "200"
s6 = s4 + s5
print()
print(s6)
```

```
 ----jGRASP exec: python StringOperators01.py

Argentine Tango

100200

 ----jGRASP: operation complete.
```

Program **StringOperators02.py**, shown in Figure 14.4, reviews *string concatenation* with the *overloaded* plus equals (**+=**) operator. Remember, the reason **+** and **+=** are called "overloaded operators" is that they can perform more than one job (as in mathematical addition and string concatenation).

Figure 14.4

```python
# StringOperators02.py
# This program demonstrates that the <+=> operator
# is also overloaded and can be used to concatenate
# one string to the end of another.


s1 = "Argentine "
s2 = "Tango"
s1 += s2
print()
print(s1)

s4 = "100"
s5 = "200"
s4 += s5
print()
print(s4)
```

```
 ----jGRASP exec: python StringOperators02.py

Argentine Tango

100200

 ----jGRASP: operation complete.
```

What Exactly Are Strings?

Before we move on to the next program, consider Figure 14.5. This shows a representation of a string with a **length** of **5**. You will note that not only do strings and arrays share the concept of “length”, they also share the concept of *indexes*. You will also note that just like arrays, the first index is **0** and the last index is the **length** minus **1**.

Figure 14.5

0	1	2	3	4
T	E	X	A	S

Based on the similarities, you might begin to wonder if a string actually is an array of characters. If you are programming in Java, the answer is “No”. If you were taking AP® Computer Science-A back in the 1980s, when we taught Standard Pascal, which had no string data type, we had no choice but to use an array of characters. What about Python? Is a string an array of characters in Python? Program **StringFunctions04.py**, in Figure 14.9, investigates this. We have string variable **state** which stores the string literal **"TEXAS"**. We want to display the character at index **2**. If we look back at Figure 14.8, we can tell this should be an **'X'**. The program tries to access this character by using the index operator or brackets **[ ]** just like an array. Does this work? Yes, it does.

Figure 14.6

```
# StringOperators03.py
# This program demonstrates that the index operator []
# can be used to access the individual characters in a
# string, as if a string were an array of characters.
# NOTE: Like arrays, string indexes also start with 0.


state = "TEXAS"
print()
print(state[2])
```

```
 ----jGRASP exec: python StringOperators03.py

X

 ----jGRASP: operation complete.
```

<u>Working With Substrings</u>

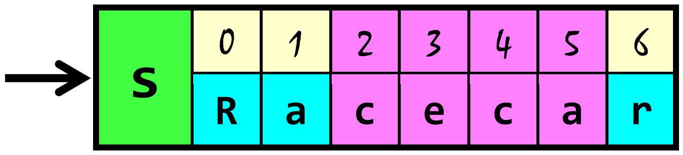
Sometimes we need to isolate a piece of a string, or a *substring*. For example: We could be storing someone's full name and then need to extract their first name and their last name. We also could be storing an address and need to extract their City, their State and their Zip Code. Working with sub strings is a major part of *String Processing*.

Program **StringOperators04.py**, shown in Figure 14.6, demonstrates how to use the isolate a substring. When you look at the code, it may look familiar; as in it is the exact same syntax that we used when *slicing* arrays in the previous chapter. In Python, isolating substrings is done via *string slicing*.

Remember, when you look at those 2 numbers inside the brackets, the substring will start at the index of the first number and stop <u>before</u> the second. If there is no first number, it starts at the beginning of the string. If there is no second number, it goes to the end of the string.

Figure 14.6

```
# StringOperators04.py
# This program demonstrates how to access
# a piece of a string.
# This is usually called a "sub-string".
# It is accomplished via "String Slicing"
# which is very similar to "Array Slicing".


s = "Racecar"

print()
print(s[0:4])
print(s[1:4])
print(s[2:4])
print(s[2:6])
print(s[3:6])
print(s[4:7])
print(s[4:])
print(s[:4])

```



```
 ----jGRASP exec: python StringOperators04.py

Race
ace
ce
ceca
eca
car
car
Race

 ----jGRASP: operation complete.
```

String *Multiplication*?

This may sound weird, but you did see *Array Multiplication* in the previous chapter. *String Multiplication* is very similar and it provides an efficient tool when you need to concatenate a string to itself multiple times.

Program **StringOperators05.py**, shown in Figure 14.7, demonstrates how the overload asterisk operator (*****) is used to "multiply" strings. This is very similar to the way the same operator is used to multiply arrays.

Figure 14.7

```python
# StringOperators05.py
# This program demonstrates that you can
# actually "multiply" a string by an integer
# with the overloaded <*> operator.
# NOTE: This is very similar to
#       "Array Multiplication".


s1 = "Racecar"
s2 = s1 * 3

print()
print(s1)
print(s2)
```

```
 ----jGRASP exec: python StringOperators05.py

Racecar
RacecarRacecarRacecar

 ----jGRASP: operation complete.
```

NOTE: When using the terms “Array Multiplication” and “String Multiplication”, it needs to be understood that Arrays and Strings can only be “multiplied” by integer values. You cannot “multiply” an array by another array or a string by another string.

Comparing Strings

For *String Processing* to be affective, we need to be able to compare strings. Sometimes we need to see if 2 strings match. Sometimes we need to see if one string alphabetically goes before the other.

Program **StringOperators06.py**, in Figure 14.8, demonstrates that the double equals operator **==**, which is used to compare for equality with simple data types, also works with strings.

Figure 14.8

```
# StringOperators06.py
# This program demonstrates the "is equal to"
# operator == can be used to compare 2 strings
# for equality.


s1 = "Foxtrot"
s2 = "Waltz"
s3 = "Foxtrot"

print()
print(s1 == s2)
print(s1 == s3)

```

```
 ----jGRASP exec: python StringOperators06.py

False
True

 ----jGRASP: operation complete.
```

Program **StringOperators07.py**, shown in Figure 14.9, demonstrates that the greater than operator **>** and the less than operator **<** can be used when comparing strings to see which goes alphabetically before the other. If **s1** < **s2** is **True** then **s1** goes before **s2**. If **s1** > **s2** is **True** then **s1** goes after **s2**. If neither is **True**, then **s1** and **s2** must be equal. This program has multiple executions to test all of the different possibilities.

Figure 14.9

```
# StringOperators07.py
# This program demonstrates that the "greater than" > and
# less than < operators can compare strings alphabetically.
# NOTE: This program will not work properly if one string
# starts with a CAPITAL letter and the other string does not.
# A later program example will fix this.


print()
s1 = input("Enter 1st string.  -->  ")
s2 = input("Enter 2nd string.  -->  ")
print()

if s1 < s2:
   print(s1,"goes alphabetically before",s2)
elif s1 > s2:
   print(s1,"goes alphabetically after",s2)
else:
   print("Both strings are equal")

```

```
  ----jGRASP exec: python StringOperators07.py

▶▶ Enter 1st string.  -->  NEON
▶▶ Enter 2nd string.  -->  ZEBRA

   NEON goes alphabetically before ZEBRA

  ----jGRASP: operation complete.
```

```
  ----jGRASP exec: python StringOperators07.py

▶▶ Enter 1st string.  -->  banana
▶▶ Enter 2nd string.  -->  apple

   banana goes alphabetically after apple

  ----jGRASP: operation complete.
```

```
  ----jGRASP exec: python StringOperators07.py

▶▶ Enter 1st string.  -->  Computer
▶▶ Enter 2nd string.  -->  Computer

   Both strings are equal

  ----jGRASP: operation complete.
```

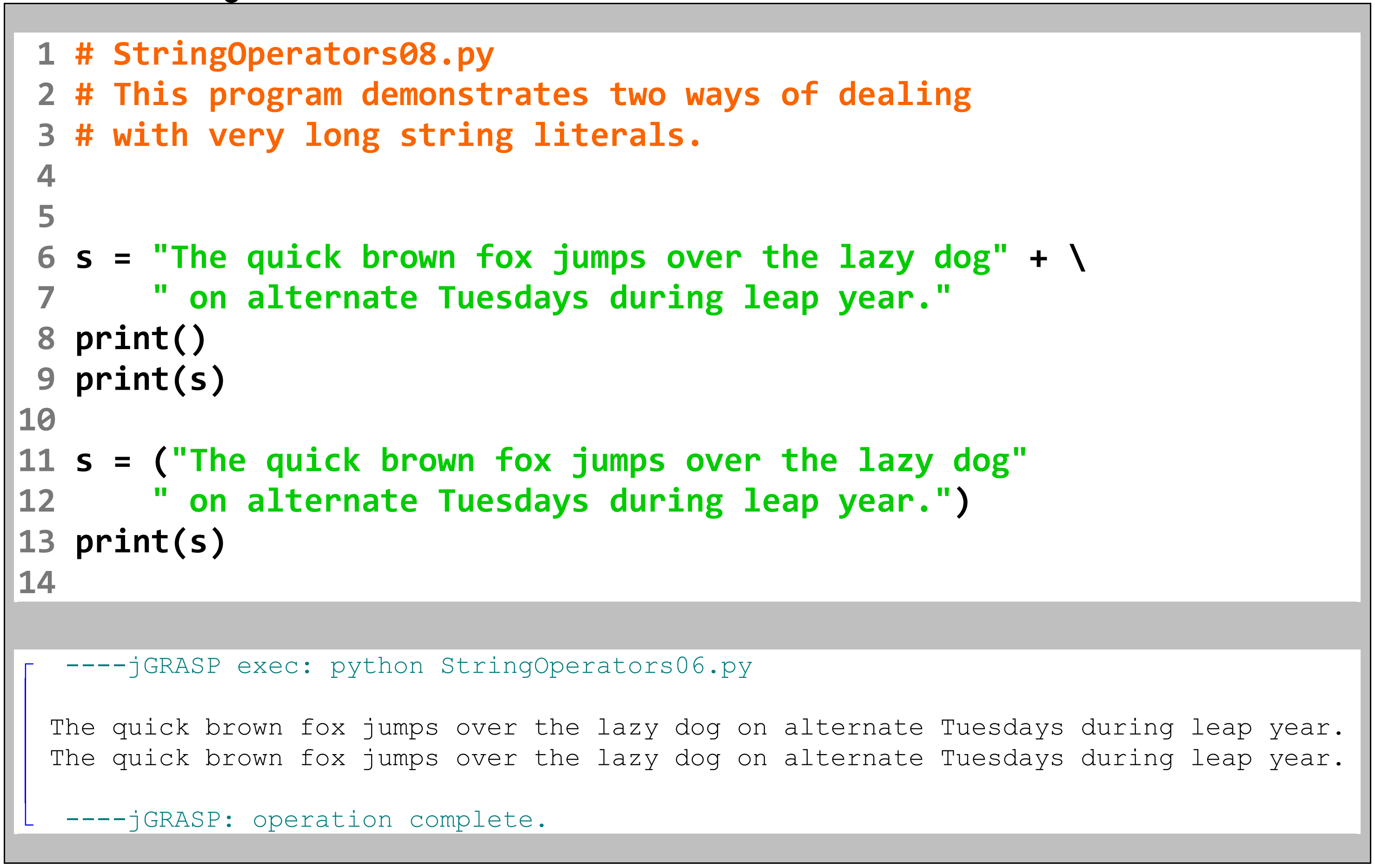
```
  ----jGRASP exec: python StringOperators07.py

▶▶ Enter 1st string.  -->  apple
▶▶ Enter 2nd string.  -->  ZEBRA

   apple goes alphabetically after ZEBRA

  ----jGRASP: operation complete.
```

NOTE: The issue with comparing strings with different cases was first introduced in the previous chapter. Later in this chapter the issue will finally be resolved.

Multiline Strings and Really Long Strings

Sometimes, you have a string literal that is really long. While IDEs like **jGRASP** do not mind long string literals, it can make your program harder to read because you only see part of the string. Program **StringOperators08.py**, in Figure 14.10, shows a couple ways to deal with this and make the entire string literal visible on the screen.

Figure 14.10



```python
# StringOperators08.py
# This program demonstrates two ways of dealing
# with very long string literals.


s = "The quick brown fox jumps over the lazy dog" + \
    " on alternate Tuesdays during leap year."
print()
print(s)

s = ("The quick brown fox jumps over the lazy dog"
    " on alternate Tuesdays during leap year.")
print(s)

```

```
 ----jGRASP exec: python StringOperators06.py

The quick brown fox jumps over the lazy dog on alternate Tuesdays during leap year.
The quick brown fox jumps over the lazy dog on alternate Tuesdays during leap year.

 ----jGRASP: operation complete.
```

Program **StringOperators09.py**, in Figure 14.11, demonstrates an operator that you have technically seen before. The *triple-double quote* (**"""**) which we use for multiline comments is actually meant for *multiline string literals*. There are 2 examples of multiline string literals in this program. While you may prefer the appearance of the first, the output shows that all of the blank space on the left is part of the multiline string. The second example is actually more proper.

Figure 14.11

```python
# StringOperators09.py
# This program demonstrates printing a
# multi-line string literal.


print()
print("""The quick
         brown fox
         jumps over
         the lazy dog.""")

print()
print("""The quick
brown fox
jumps over
the lazy dog.""")

```

```
 ┌  ----jGRASP exec: python StringOperators09.py

   The quick
            brown fox
            jumps over
            the lazy dog.

   The quick
   brown fox
   jumps over
   the lazy dog.

 └  ----jGRASP: operation complete.
```

Program **StringOperators10.py**, in Figure 14.12, finishes this section by demonstrating a multiline string literal stored in string variable.

Figure 14.12

```python
# StringOperators10.py
# This program demonstrates storing a
# multi-line string literal in a string
# variable and then printing it.


s = """The quick
brown fox
jumps over
the lazy dog."""

print()
print(s)

```

```
 ----jGRASP exec: python StringOperators10.py

The quick
brown fox
jumps over
the lazy dog.

 ----jGRASP: operation complete.
```

14.3 String Commands

As with arrays, there are a number of commands that can be used to help with *String Processing*. Some of these should look familiar.

Program **StringCommands01.py**, in Figure 14.13, demonstrates the **len** function. You may be experiencing some déjà vu here. Yes, you have seen this before. **len** is a function that work with both arrays and strings. With array, **len** returns the number of items in the array. With strings, **len** returns the number of characters in the string. Note that I said "characters" and not "letters". All characters are counted. This means letters, digits, symbols, and even invisible characters like spaces.

Figure 14.13

```
# StringCommands01.py
# This program demonstrates the <len> command.
# In the same way that <len> will tell you
# how many items are in an array, it will also
# tell you how many characters are in a string.


s1 = "Argentine"
s2 = "Tango"
s3 = s1 + " " + s2

print()
print(s1,"has",len(s1),"characters.")
print(s2,"has",len(s2),"characters.")
print(s3,"has",len(s3),"characters.")

```

```
 ----jGRASP exec: python StringCommands01.py

Argentine has 9 characters.
Tango has 5 characters.
Argentine Tango has 15 characters.

 ----jGRASP: operation complete.
```

Program **StringCommands02.py**, in Figure 14.14, uses the **len** function to demonstrate why I am not a big fan of multiline string literals. When you look at the program, it seems like **s1** and **s2** are storing the exact same multiline string literal. You would expect that the **len**gths of both of these would be identical. When you run the program, you see this is not the case. While **s1** has a **len**gth of **44**, **s2** has a **len**gth of **171**. What?

Figure 14.13

```python
# StringCommands02.py
# This program uses the <len> command to demonstrate
# a potential problem with multiline string literals.


s1 = """The quick
brown fox
jumps over
the lazy dog."""

s2 = """The quick
brown fox
jumps over
the lazy dog."""

print()
print("s1 has",len(s1),"characters.")
print("s2 has",len(s2),"characters.")
```

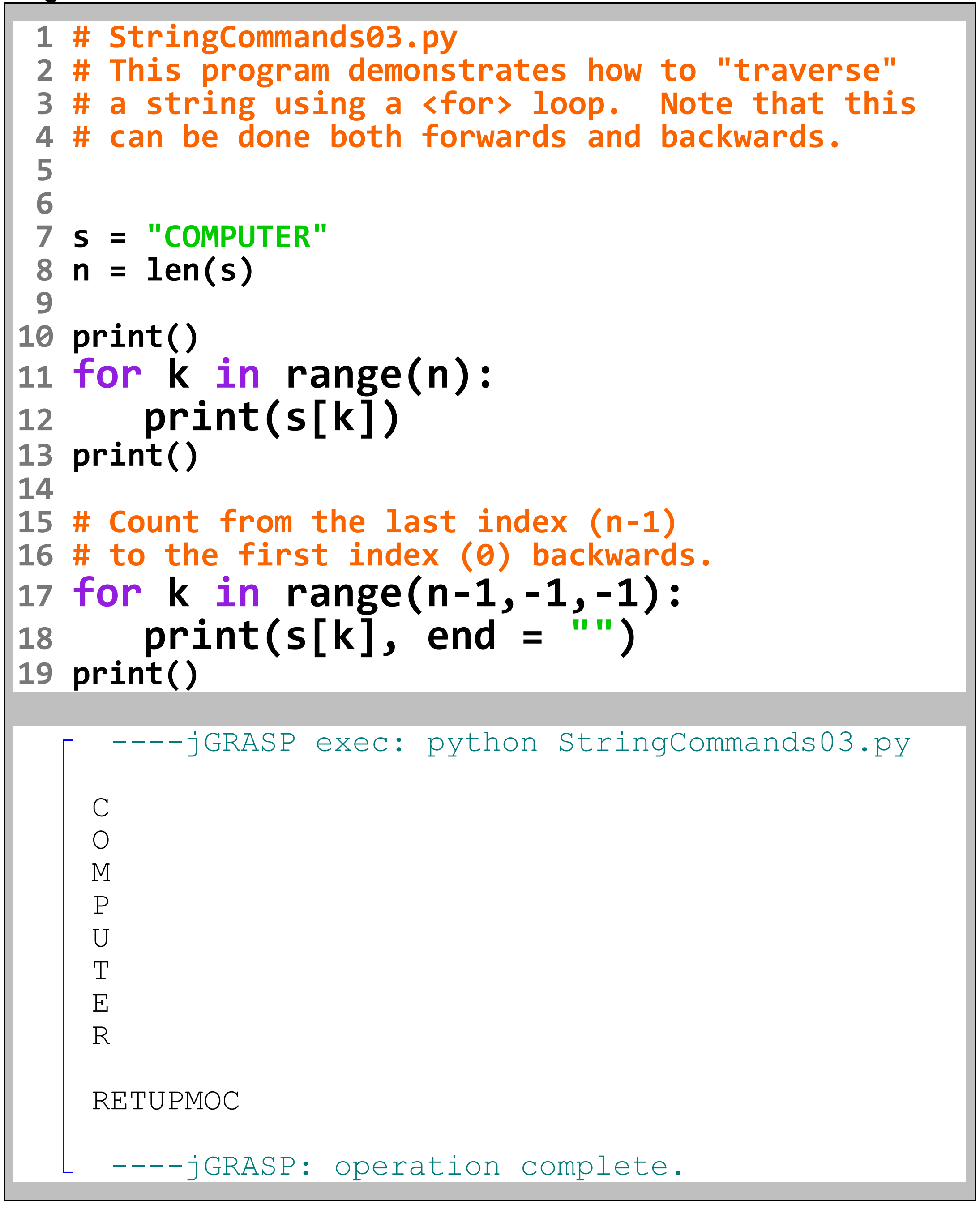
```
  ----jGRASP exec: python StringCommands02.py

 s1 has 44 characters.
 s2 has 171 characters.

  ----jGRASP: operation complete.
```

What causes this weird output is the fact that **s1**'s string literal has no spaces on the right side; however, there are many, unseen, spaces on the right side of **s2**'s string literal.

Traversing and Reversing Strings

We can *traverse* a string, character by character, in the same way that we traverse an array using a trusty **for** loop. Program **StringCommands03.py**, in Figure 14.15, demonstrates this using 2 different **for** loops. The first traverses "forwards" (from the first character to the last) and displays each character in the string on a separate line. The second loop traverses "backwards" (from the last character to the first) and displays each character in the string on the same line.

Figure 14.15



```python
# StringCommands03.py
# This program demonstrates how to "traverse"
# a string using a <for> loop.  Note that this
# can be done both forwards and backwards.


s = "COMPUTER"
n = len(s)

print()
for k in range(n):
    print(s[k])
print()

# Count from the last index (n-1)
# to the first index (0) backwards.
for k in range(n-1,-1,-1):
    print(s[k], end = "")
print()
```

```
 ----jGRASP exec: python StringCommands03.py

C
O
M
P
U
T
E
R

RETUPMOC

 ----jGRASP: operation complete.
```

At first, program **StringCommands04.py**, in Figure 14.16, looks like it is repeating the second part of the previous program. Actually, what this program is doing is different. The previous program took a string and displayed it in reverse order. This program will take one string and use it to create a second string, which will be the reverse of the first.

Figure 14.16

```python
# StringCommands04.py
# This program demonstrates how to take
# one string and create a second string
# that is a reverse of the first.


s1 = "Madam I'm Adam"
s2 = ""
n = len(s1)

# Count from the last index (n-1)
# to the first index (0) backwards.
for k in range(n-1,-1,-1):
   s2 += s1[k]

print()
print(s1)
print(s2)

```

```
 ----jGRASP exec: python StringCommands04.py

Madam I'm Adam
madA m'I madaM

 ----jGRASP: operation complete.
```

NOTE: You should refer back to this program when you do Lab 14A or 14B.

Some of you may remember that there is a **reverse** command that can be used to "reverse" an array. Would it work with strings? It seems logical, especially since both use the **len** function. It also would be a much simpler way of doing the previous program. Program **StringCommands05.py**, in Figure 14.17, investigates this by trying to use the **reverse** command with a string. Alas. this does not work. Some array commands also work with strings, but not all of them.

Figure 14.17

```
# StringCommands05.py
# This program demonstrates that
# <reverse> does not work with strings.
# NOTE: Many array commands work with
# strings, but not all of them.


s1 = "Madam I'm Adam"
s2 = s1.reverse()

print()
print(s1)
print(s2)

```

```
 ----jGRASP exec: python StringCommands05.py
Traceback (most recent call last):
  File "StringCommands05.py", line 9, in <module>
    s2 = s1.reverse()
AttributeError: 'str' object has no attribute 'reverse'

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

14.4 More String Commands

The previous section showed just enough “string commands” for you to get started on Lab 14. This section shows the remainder of the string commands.

Finding and Replacing Substrings

We talked about *substrings* earlier in the chapter. There are times when we need to look for a substring inside another, larger string. This can be doing with the **find** and **rfind** (*reverse find*) commands. The difference between the 2 is that when **find** searches for a substring, it starts at the beginning of the string and searches forwards. **rfind** starts at the end of the string and searches backwards.

Program **StringCommands06.py**, in Figure 14.18, demonstrates both commands. Note that when a substring, like **"car"** is in a larger string, like **"racecar in the carport"** multiple times, **find** will return the index of the first occurrence of the substring and **rfind** will return the index of the last. It should make sense that if a substring is only in the larger string once, like **"car"** in **"racecar"**, that the index of the first occurrence and the last occurrence of the substring is the same. If the substring is not in the other string at all, like looking for **"qwerty"** inside **"car"**, both functions will return **-1**, which is not a possible index. Technically, you can use **index** and **rindex** as well; however, **find** and **rfind** are preferable because the other 2 commands crash when the substring is not found.

Figure 14.18

```python
# StringCommands06.py
# This program demonstrates the <find> and "Reverse Find" <rfind> commands.
# <find> returns the index of the first occurrence of the substring parameter.
# <rfind> returns the index of the last occurrence of the substring parameter.
# Both return -1 if the substring parameter is not found in the original string.


s1 = "racecar"
s2 = "racecar in the car port"
s3 = "car"

index1 = s1.find(s3)
index2 = s1.rfind(s3)
index3 = s2.find(s3)
index4 = s2.rfind(s3)
index5 = s3.find("qwerty")
```

```
index6 = s3.rfind("qwerty")

print()
print("The first occurrence of",s3,"in",s1,"is at index",index1)
print("and the last occurrence is at index",index2)

print()
print("The first occurrence of",s3,"in",s2,"is at index",index3)
print("and the last occurrence is at index",index4)

print()
print("The first occurrence of qwerty in",s3,"is at index",index5)
print("and the last occurrence is at index",index6)
```

```
 ----jGRASP exec: python StringCommands06.py

The first occurrence of car in racecar is at index 4
and the last occurrence is at index 4

The first occurrence of car in racecar in the car port is at index 4
and the last occurrence is at index 15

The first occurrence of qwerty in car is at index -1
and the last occurrence is at index -1

 ----jGRASP: operation complete.
```

Program **StringCommands07.py**, in Figure 14.19, demonstrates the **count**, **startswith** and **endswith** commands. **count** is similar to **find**, but instead of looking for only the first occurrence of a substring in a larger string, it finds them all and counts them. As for **startswith**, it will return **True** if the string "starts with" a particular substring. If not, it will return **False**. The **endswith** command works the same way at the end of the string.

Figure 14.19

```
# StringCommands07.py
# This program demonstrates the <count>,
# <startswith> and <endswith> commands.


s = "HOW MANY BANANAS ARE ON ANA'S BANANA BOAT"
```

```
print()
print(s.count("BANANA"))
print(s.count("AN"))
print(s.count("AN",9,29))
print(s.startswith("HOW"))
print(s.startswith("BANANA"))
print(s.endswith("BOAT"))
print(s.endswith("boat"))
```

```
 ----jGRASP exec: python StringCommands07.py

2
6
3
True
False
True
False

 ----jGRASP: operation complete.
```

Sometimes, when you *find* a substring; you want to *replace* it with another. I am sure that you have used the *Find and Replace* feature on your word processor at some point in your life. Program **StringCommands08.py**, in Figure 14.20, demonstrates the **replace** command which can "find and replace" individual characters or multi-character substrings.

Figure 14.20

```
# StringCommands08.py
# This program demonstrates the <replace> command
# which can replace characters or substrings.
# NOTE: The <replace> command returns an altered
#       COPY of the original string. It does not
#       alter the original string in ANY way.


s1 = "racecar"
s2 = s1.replace('r','l')
s3 = s1.replace("ace","eally awesome ")

```

```
print()
print(s1)
print(s2)
print(s1)
print(s3)
```

```
 ----jGRASP exec: python StringCommands08.py

racecar
lacecal
racecar
really awesome car

 ----jGRASP: operation complete.
```

Look closely at the previous program and its output. Do you see that the **replace** commands come <u>before</u> the **print** commands? Even so, note that whenever **s1** is displayed, we see the original value of "**racecar**". There is something very important that you need to realize about string commands. They do <u>not</u> alter the original string. Some of them, like **replace**, create a <u>COPY</u> of the string which is altered, and that copy is returned. So **s2** received a copy of **s1** where both occurrences of **'r'** were replaced with **'l'** and **s2** received a different copy of **s1** where **"ace"** was replaced with **"eally awesome"**.

Question, what do you do if you actually want to alter the original string variable? The answer is demonstrated in program **StringCommands09.py**, shown in Figure 14.21. The secret is to take the value that is returned and assign it back to the original variable.

Figure 14.21

```
# StringCommands09.py
# This program demonstrates what you need to do to replace
# characters/substrings in the original string.


s1 = "racecar"
s1 = s1.replace('r','l')

s2 = "racecar"
s2 = s2.replace("ace","eally awesome ")

```

```
print()
print(s1)
print(s2)
```

```
 ----jGRASP exec: python StringCommands09.py

lacecal
really awesome car

 ----jGRASP: operation complete.
```

Converting Strings

Since the last chapter, we have had issues with strings when some start with a capital letter and others start with a lowercase letter. This messes up comparisons, which in turn messes up sorting. The solution to this problem is to convert all of the strings to either all UPPERCASE or to all lowercase. Either one will work.

Program **StringCommands10.py**, in Figure 14.22, demonstrates the **upper** and **lower** commands which will convert a string to all UPPERCASE and all lowercase respectively. Now if your string has a mixture of CAPITAL letters, lowercase letters, digits, and symbols, the **upper** function will only affect the lowercase letters in the string. Likewise, the **lower** function will only affect the UPPERCASE letters in the string.

Figure 14.22

```
# StringCommands10.py
# This program demonstrates the <upper> and <lower> commands
# which convert string to all UPPERCASE or all lowercase.
# NOTE: Like the <replace> command, <upper> and <lower>
#       do not alter the original string in ANY way.


s1 = "racecar"
s2 = "RaCeCaR"
s3 = ">RACECAR-100<"

```

```python
print("\nUppercased Strings:")
print(s1.upper())
print(s2.upper())
print(s3.upper())

print("\nLowercased Strings:")
print(s1.lower())
print(s2.lower())
print(s3.lower())

print("\nOriginal Strings:")
print(s1)
print(s2)
print(s3)
```

```
 ----jGRASP exec: python StringCommands10.py

Uppercased Strings:
RACECAR
RACECAR
>RACECAR-100<

Lowercased Strings:
racecar
racecar
>racecar-100<

Original Strings:
racecar
RaCeCaR
>RACECAR-100<

 ----jGRASP: operation complete.
```

Look closely at this output. Note that when the original strings are displayed at the end of the program, they still have their original values. This is because the **upper** and **lower** functions, like **replace**, do not alter the original string at all. Instead, they return an altered COPY of the string, Program **StringCommands11.py**, in Figure 14.23, demonstrates what to need to do to essentially UPPERCASE or lowercase the original string variable's contents. As before, the secret is to assign the return value back to the original string.

Figure 14.23

```python
# StringCommands11.py
# This program demonstrates how to UPPERCASE
# or lowercase the original string.


s = ">RaCeCaR<"

print()
print(s)

s = s.upper()
print(s)

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

```
 ----jGRASP exec: python StringCommands11.py

>RaCeCaR<
>RACECAR<
>racecar<

 ----jGRASP: operation complete.
```

Now that we understand the **upper** and **lower** functions, we can finally solve the UPPERCASE/lowercase string comparison issue. Program **StringCommands12.py**, in Figure 14.24, is very similar to program **StringOperators07.py** from earlier in the chapter. As before, this program will allow the user to enter 2 strings and then it will tell you if the first string is alphabetically "before" or "after" the second. This difference is in line 12. Instead of comparing the original strings **s1** and **s2**, we are comparing the **upper**cased versions of the strings with **s1.upper()** and **s2.upper()**. The result is the program now behaves the way we want it to. It should be noted that the program would work just as well if you used function **lower** instead of **upper**.

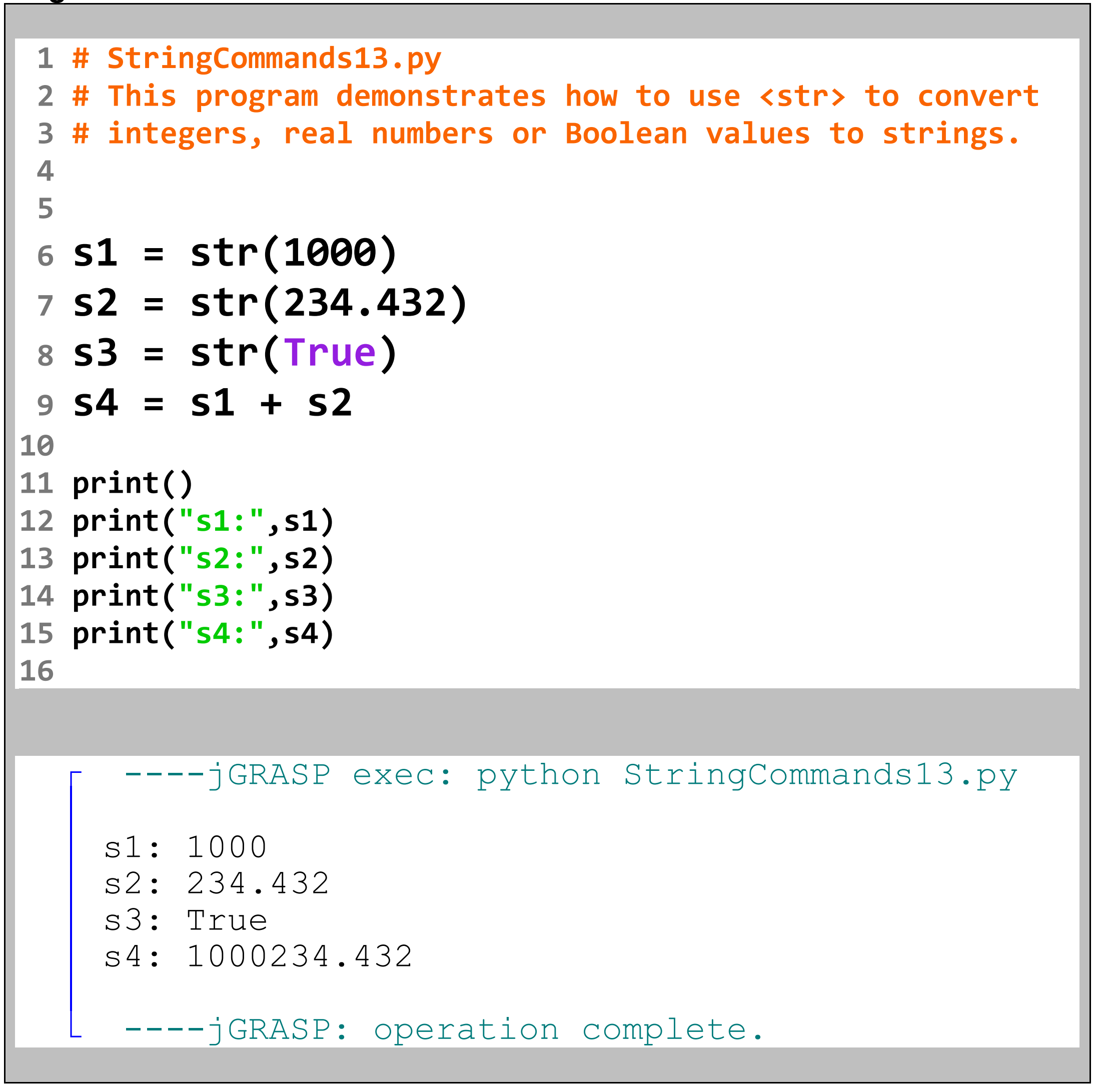
Figure 14.24

```python
# StringCommands12.py
# This program fixes the "case" issue of StringOperators07.py
# by using the <upper> command.  With both strings converted
# to UPPERCASE, the strings can be compared properly.
# NOTE: Using the <lower> command would also work.

print()
s1 = input("Enter 1st string.  -->  ")
s2 = input("Enter 2nd string.  -->  ")
print()

if s1.upper() < s2.upper():
   print(s1,"goes alphabetically before",s2)
elif s1.upper() > s2.upper():
   print(s1,"goes alphabetically after",s2)
else:
   print("Both strings are equal")
```

```
 ----jGRASP exec: python StringCommands12.py

Enter 1st string.  -->  apple
Enter 2nd string.  -->  ZEBRA

apple goes alphabetically before ZEBRA

 ----jGRASP: operation complete.
```

When is come to converting strings, it is not just about converting between cases. It is also about converting between *data types*. Sometimes you need to convert a number to a string. Sometimes you need to convert a string to a number. Program **StringCommands13.py**, in Figure 14.25, reviews how to convert numbers, both integers and real numbers, as well and Boolean values to strings with the **str** function. You actually first saw this way back in chapter 6. To prove that the conversion actually took place the 2 “numbers” are “added”. When that value is displayed, we see that we did not get numeric addition, we got string concatenation.

Figure 14.25



```
# StringCommands13.py
# This program demonstrates how to use <str> to convert
# integers, real numbers or Boolean values to strings.


s1 = str(1000)
s2 = str(234.432)
s3 = str(True)
s4 = s1 + s2

print()
print("s1:",s1)
print("s2:",s2)
print("s3:",s3)
print("s4:",s4)

```

```
----jGRASP exec: python StringCommands13.py

s1: 1000
s2: 234.432
s3: True
s4: 1000234.432

----jGRASP: operation complete.
```

Program **StringCommands14.py**, in Figure 14.26, demonstrates how to convert the other way, from string to some other data type like integer, real number or Boolean. These conversion are done with the **int**, **float** and **bool** functions. It should make sense that **int** is for integer and **bool** is for Boolean. If you are wondering about **float**, remember that another term for a “real number” is a “floating point number”. To prove that the conversions were successful the 2 converted numbers are added. This time we do get numeric addition. Also, the Boolean value is used in an **if** statement.

Figure 14.26

```
# StringCommands14.py
# This program demonstrates how to use <int>, <float> and
# <bool> to convert string values to integers, real numbers
# (a.k.a. floating point numbers) and Boolean values.


n1 = int("1000")
n2 = float("234.432")
n3 = n1 + n2
b1 = bool("True")

print()
print("n1:",n1)
print("n2:",n2)
print("n3:",n3)

if b1:
   print("b1:",b1)

```

```
 ----jGRASP exec: python StringCommands14.py

n1: 1000
n2: 234.432
n3: 1234.432
b1: True

 ----jGRASP: operation complete.
```

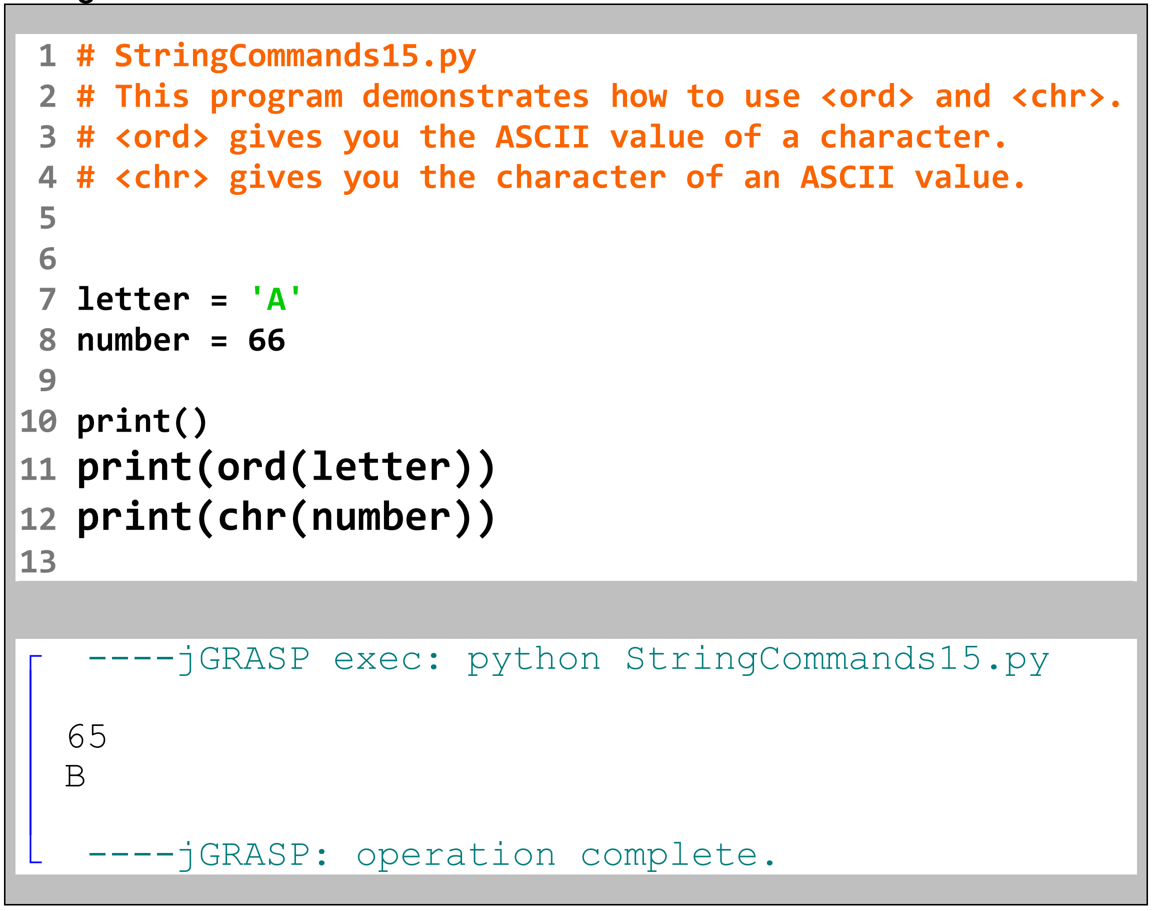
We are now going to look at 2 commands that are often used together in the same program. These are **ord** and **chr**. The **ord** command will take a character and return its ASCII value. The **chr** command does the reverse, it takes an integer ASCII number and returns its assigned character. At first it may seem like **ord** and **chr** are just like **int** and **str**, but they are actually very different. The chart in Figure 14.27 shows just how different. The **int** command only works if the string contains nothing by numeric digits. If that is the case, **int** essentially removes the quotes. On the other hand, **ord** can only work with single characters. **chr** only works with integers between 0 and 255 and **str** basically works with everything.

Figure 14.27

`int('1') = 1`	`ord('1') = 49`
`str(65) = "65"`	`chr(65) = 'A'`
`int('A') = Error`	`ord('A') = 65`
`str(300) = "300"`	`chr(300) = Error`
`int("65") = 65`	`ord("65") = Error`

Program **StringCommands15.py**, in Figure 14.28, demonstrates the **ord** and **chr** commands. Note how **ord('A')** returns **65** and **chr(66)** returns **B**.

Figure 14.28



```python
# StringCommands15.py
# This program demonstrates how to use <ord> and <chr>.
# <ord> gives you the ASCII value of a character.
# <chr> gives you the character of an ASCII value.


letter = 'A'
number = 66

print()
print(ord(letter))
print(chr(number))

```

```
 ----jGRASP exec: python StringCommands15.py

65
B

 ----jGRASP: operation complete.
```

While most languages have commands like **ord** and **chr**, not all languages have commands like **upper** and **lower**. If you are programming in one of those languages, you need to write your own code to “uppercase” or “lowercase” a string. This is definitely doable with the **ord** and **chr** commands. Consider this:

The ASCII value of **'A'** is **65** and the ASCII value of **'a'** is **97**.	**97** – **65** = **32**
The ASCII value of **'B'** is **66** and the ASCII value of **'b'** is **98**.	**98** – **66** = **32**
: : : : : : :	: : :
The ASCII value of **'Z'** is **90** and the ASCII value of **'z'** is **122**.	**122** – **90** = **32**

What we see here is that the difference between the ASCII values of a CAPITAL letter and its corresponding lowercase letter is always **32**. So if we have a lowercase letter, we can use **ord** to get its ASCII value. Then we subtract **32**. Then we use **chr** with the difference and we will have the correct CAPITAL letter.

Program **StringCommands16.py**, in Figure 14.29, attempts to implement this process with the user-defined **allCAPS** function – which is meant to do the exact same thing as the built-in **upper** function.

Figure 14.29

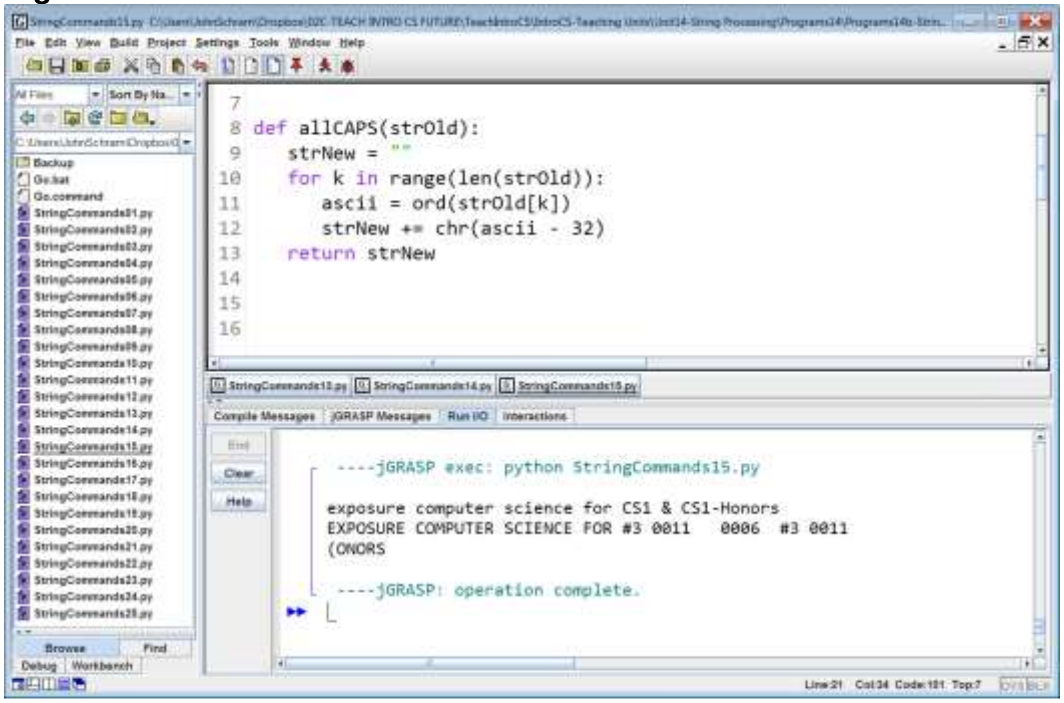
```
# StringCommands16.py
# This program attempts to use <ord> and <chr> to convert
# a string to all CAPITAL letters, like the <upper> command.
# The problem is it performs the conversion on all of the
# characters, not just the lowercase letters.


def allCAPS(strOld):
   strNew = ""
   for k in range(len(strOld)):
      ascii = ord(strOld[k])
      strNew += chr(ascii - 32)
   return strNew



##########
## MAIN ##
##########

```

```
title = "exposure computer science for CS1 & CS1-Honors"
newTitle = allCAPS(title)
print()
print(title)
print(newTitle)

```

```
 ----jGRASP exec: python StringCommands16.py

exposure computer science for CS1 & CS1-Honors
EXPOSURE COMPUTER SCIENCE FOR #3 0011   0006  #3 0011
(ONORS

 ----jGRASP: operation complete.
```

At first, out **allCAPS** functions seems to work… until we get to the first letter that is already a capital letter. When CAPITAL **C** gets "converted", it becomes a hashtag (**#**). CAPITAL **S** *converts* to a **3**. CAPITAL **H** converts to **(**. The **1** and the **&** convert to characters that cannot be displayed properly, so we see their hex codes instead. Finally, the dash (**-**) converts to the *new line character* (**\n**). It may look like the **(ONORS** is on the next line because it word wrapped. The screen shot in Figure 14.30 shows you this is not the case.

Figure 14.30



```
def allCAPS(strOld):
   strNew = ""
   for k in range(len(strOld)):
      ascii = ord(strOld[k])
      strNew += chr(ascii - 32)
   return strNew
```

```
 ----jGRASP exec: python StringCommands15.py

exposure computer science for CS1 & CS1-Honors
EXPOSURE COMPUTER SCIENCE FOR #3 0011   0006  #3 0011
(ONORS

 ----jGRASP: operation complete.
```

Program **StringCommands17.py**, in Figure 14.31, fixes the issue of the previous program by using an improved **allCAPS** function. This time, every character is first checked to see if it is a lowercase letter (between **'a'** and **'z'**). If it is, it is capitalized and added to the new string. If not, it is added to the new string without capitalizing.

Figure 14.31

```python
# StringCommands17.py
# This program fixes the issue of the previous program
# by first checking if a character is a lowercase letter
# before capitalizing it.


def allCAPS(strOld):
   strNew = ""
   for k in range(len(strOld)):
      if strOld[k] >= 'a' and strOld[k] <= 'z':
         ascii = ord(strOld[k])
         strNew += chr(ascii - 32)
      else:
         strNew += strOld[k]
   return strNew



##########
## MAIN ##
##########

title = "exposure computer science for CS1 & CS1-Honors"
newTitle = allCAPS(title)
print()
print(title)
print(newTitle)
```

```
 ----jGRASP exec: python StringCommands17.py

exposure computer science for CS1 & CS1-Honors
EXPOSURE COMPUTER SCIENCE FOR CS1 & CS1-HONORS

 ----jGRASP: operation complete.
```

The "is" Commands

There are several string commands that begin with "**is**" in Python. Program **StringCommands18.py**, in Figure 14.32, demonstrates the first 3 which are **islower**, **isupper** and **istitle**. The program also demonstrates the **capitalize** and **title** commands. Make sure you do not confuse these with **upper**. As you have seen, **upper** capitalizes every lowercase letter in a string. The **capitalize** command "capitalizes" only the first character. The **title** command capitalizes the first character of every word.

Now let us focus on those "**is**" commands.
islower returns **True** if all of the letters in a string are lowercase; **False** otherwise.
isupper returns **True** if all of the letters in a string are UPPER; **False** otherwise.
istitle returns **True** if the string is in *Title Case*; **False** otherwise. "Title Case" means the first letter of every word is capitalized and the rest are lowercase.

Figure 14.32

```
# StringCommands18.py
# This program demonstrates <capitalize>, <title>,
# <islower>, <isupper> and <istitle>.
# Note that <capitalize> is not the same thing as <upper>.


s = "exposure computer science 2020 in python for CS1 & CS1-honors"
print("\n" + s)
print(s.islower(), s.isupper() ,s.istitle())

s = s.capitalize()
print("\n" + s)
print(s.islower(), s.isupper() ,s.istitle())

s = s.upper()
print("\n" + s)
print(s.islower(), s.isupper() ,s.istitle())

s = s.lower()
print("\n" + s)
print(s.islower(), s.isupper() ,s.istitle())

s = s.title()
print("\n" + s)
print(s.islower(), s.isupper() ,s.istitle())
```

```
 ----jGRASP exec: python StringCommands18.py

exposure computer science 2020 in python for CS1 & CS1-honors
False False False

Exposure computer science 2020 in python for cs1 & cs1-honors
False False False

EXPOSURE COMPUTER SCIENCE 2020 IN PYTHON FOR CS1 & CS1-HONORS
False True False

exposure computer science 2020 in python for cs1 & cs1-honors
True False False

Exposure Computer Science 2020 In Python For Cs1 & Cs1-Honors
False False True

 ----jGRASP: operation complete.
```

Program **StringCommands19.py**, in Figure 14.33, demonstrates 4 more "**is**" commands which are **isdecimal**, **isalpha**, **isalnum** and **isidentifier**. The details of these commands are as follows:

isdecimal returns **True** if every character in the string is a numeric digit (**0-9**); **False** otherwise.

isalpha returns **True** if every character in the string is a letter; **False** otherwise.

isalnum returns **True** if every character in the string is *alphanumeric* which means it is either a letter or a numeric digit; **False** otherwise.

isidentifier returns **True** if the string would be a proper identifier for a variable or subroutine. Identifiers needs are basically alphanumeric, but they must start with a letter and can also include the underscore **_** character; **False** otherwise.

Figure 14.33

```python
# StringCommands19.py
# This program demonstrates <isdecimal>, <isalpha>, <isalnum>
# and <isidentifier>.


s = "Exposure Computer Science 2020 in Python for CS1 & CS1-Honors"
print("\n" + s)
print(s.isdecimal(), s.isalpha(), s.isalnum(), s.isidentifier())

s = "ExposureComputerScience"
print("\n" + s)
print(s.isdecimal(), s.isalpha(), s.isalnum(), s.isidentifier())

s = "1996"
print("\n" + s)
print(s.isdecimal(), s.isalpha(), s.isalnum(), s.isidentifier())

s = "3.14159"
print("\n" + s)
print(s.isdecimal(), s.isalpha(), s.isalnum(), s.isidentifier())

s = "11A2B"
print("\n" + s)
print(s.isdecimal(), s.isalpha(), s.isalnum(), s.isidentifier())

s = "A_108"
print("\n" + s)
print(s.isdecimal(), s.isalpha(), s.isalnum(), s.isidentifier())
```

```
 ----jGRASP exec: python StringCommands19.py

Exposure Computer Science 2020 in Python for CS1 & CS1-Honors
False False False False

ExposureComputerScience
False True True True

1996
True False True False

3.14159
False False False False

11A2B
False False True False

A_108
False False False True
```

```
----jGRASP: operation complete.
```

Program **StringCommands20.py**, in Figure 14.34, demonstrates a couple more "**is**" commands that are similar to **isdecimal**. These are **isdigit** and **isnumeric**. The details of these commands are as follows:

isdigit is like **isdecimal** except it will also accept the ¹, ² and ³ characters as "digits". Please understand that this is not just the numbers 1, 2 and 3 in a superscript font. They are actual built-in characters.

isnumeric is like **isdigit** except it will also accept the built-in ¼, ½ and ¾ characters as "numeric" values.

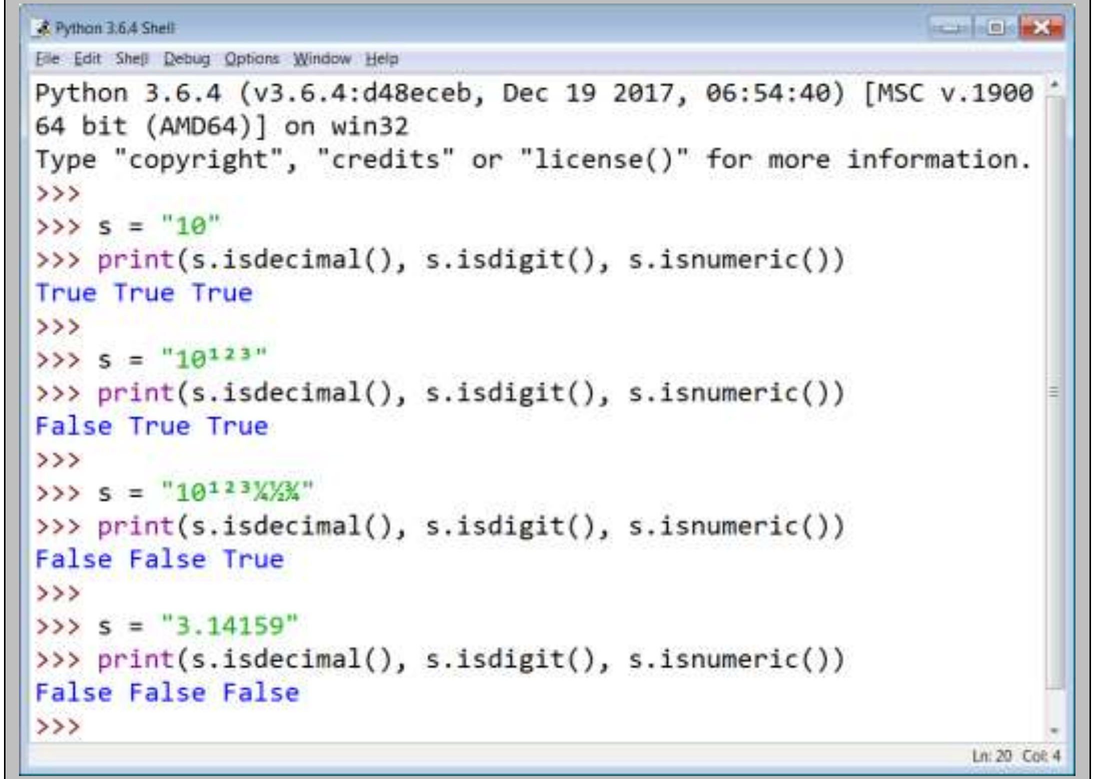
NOTE: Ironically, none of these commands will accept a decimal point.

ALSO: If you try to execute this program in **jGRASP** you will get an error because the exponents and fractions are "**non-UTF-8 code**". The program can be executed line by line from the **Python Shell**. It is not really important for you to figure out how to execute the program right now. Just look at the output that I am providing.

Figure 14.34

```
# StringCommands20.py
# This program demonstrates how <isdigit> and
# <isnumeric> are different from <isdecimal>.
# NOTE: This program will not execute in jGRASP.
# You will need to use the Python Shell.

s = "10"
print(s.isdecimal(), s.isdigit(), s.isnumeric())

s = "10¹²³"
print(s.isdecimal(), s.isdigit(), s.isnumeric())

s = "10¹²³¼½¾"
print(s.isdecimal(), s.isdigit(), s.isnumeric())

s = "3.14159"
print(s.isdecimal(), s.isdigit(), s.isnumeric())
```

```
18
```



```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900
64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> s = "10"
>>> print(s.isdecimal(), s.isdigit(), s.isnumeric())
True True True
>>>
>>> s = "10¹²³"
>>> print(s.isdecimal(), s.isdigit(), s.isnumeric())
False True True
>>>
>>> s = "10¹²³¼½¾"
>>> print(s.isdecimal(), s.isdigit(), s.isnumeric())
False False True
>>>
>>> s = "3.14159"
>>> print(s.isdecimal(), s.isdigit(), s.isnumeric())
False False False
>>>
```

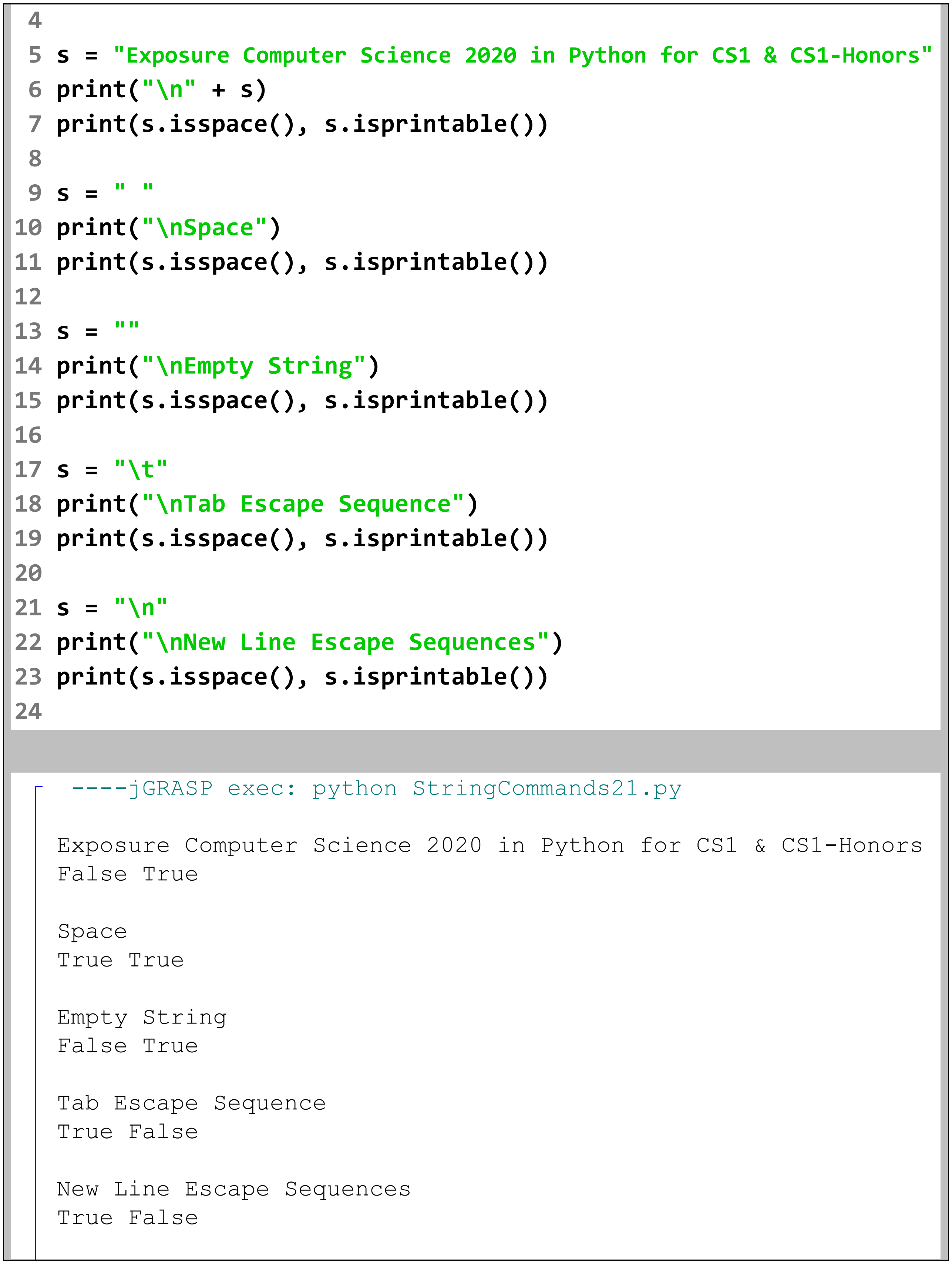
Program **StringCommands21.py**, in Figure 14.35, demonstrates the last 2 "**is**" commands. These are **isspace** and **isprintable**. The details of these commands are as follows:

isspace returns **True** if every character in the string is a valid *whitespace* character; **False** otherwise. Whitespace characters include the space character as well as other characters like the *tab character* (**`\t`**) and the *new line character* (**`\n`**).

isprintable returns **True** if every character in the string is *printable*; **False** otherwise. Any character that you can type is "printable". Empty strings are also "printable". Examples of characters that are not printable are *tab* and *new line*.

Figure 14.35

```
# StringCommands21.py
# This program demonstrates <isspace> and <isprintable>.

```



```python

s = "Exposure Computer Science 2020 in Python for CS1 & CS1-Honors"
print("\n" + s)
print(s.isspace(), s.isprintable())

s = " "
print("\nSpace")
print(s.isspace(), s.isprintable())

s = ""
print("\nEmpty String")
print(s.isspace(), s.isprintable())

s = "\t"
print("\nTab Escape Sequence")
print(s.isspace(), s.isprintable())

s = "\n"
print("\nNew Line Escape Sequences")
print(s.isspace(), s.isprintable())

```

```
 ----jGRASP exec: python StringCommands21.py

Exposure Computer Science 2020 in Python for CS1 & CS1-Honors
False True

Space
True True

Empty String
False True

Tab Escape Sequence
True False

New Line Escape Sequences
True False
```

```
 └  ----jGRASP: operation complete.
```

Breaking Up Sentences into Words

Sometimes, you have a string that holds an entire sentence and you need to break it up into its individual words. This can be done with the **split** command. Program **StringCommands22.py**, in Figure 14.36, shows how the **split** command "splits" a sentence into an array of words. By default, spaces are used as delimiters (word separators). You can also specify alternate delimiting characters or strings.

Figure 14.36

```
# StringCommands22.py
# This program demonstrates how to use the <split> command to
# "split" a something like a sentence into an array of words.


title = "Exposure Computer Science 2020 for CS1 & CS1-Honors"
words = title.split()
print()
print(words)

title = "Exposure, Computer, Science, 2020, for, CS1, &, CS1-Honors"
words = title.split()
print()
print(words)

title = "Exposure, Computer, Science, 2020, for, CS1, &, CS1-Honors"
words = title.split(',')
print()
print(words)

title = "Exposure, Computer, Science, 2020, for, CS1, &, CS1-Honors"
words = title.split(', ')
print()
print(words)

title = "Exposure Computer Science 2020 for CS1 & CS1-Honors"
words = title.split('&')
print()
print(words)

title = "Exposure Computer Science 2020 for CS1 & CS1-Honors"
```

```
words = title.split('%')
print()
print(words)
```

```
 ----jGRASP exec: python StringCommands22.py

['Exposure', 'Computer', 'Science', '2020', 'for', 'CS1', '&', 'CS1-Honors']

['Exposure,', 'Computer,', 'Science,', '2020,', 'for,', 'CS1,', '&,', 'CS1-Honors']

['Exposure', ' Computer', ' Science', ' 2020', ' for', ' CS1', ' &', ' CS1-Honors']

['Exposure', 'Computer', 'Science', '2020', 'for', 'CS1', '&', 'CS1-Honors']

['Exposure Computer Science 2020 for CS1 ', ' CS1-Honors']

['Exposure Computer Science 2020 for CS1 & CS1-Honors']

 ----jGRASP: operation complete.
```

There is another command that is essentially the opposite of **split**. This is the **join** command. While **split** "splits" a sentence into an array of words; **join** "joins" an array of words together to form a sentence. Program **StringCommands23.py**, in Figure 14.37, shows how this works. Keep in mind that when you *join* together your words to make a sentence, you need to specify something to separate the words. The first **join** does this with a **space** which is probably most common. The second **join** uses a comma (**,**) and the third uses a weird string **"<~>"**. For the last one, I did not want any separate between the words. That is why I used the empty string **""** as my separator.

Figure 14.37

```
# StringCommands23.py
# This program demonstrates how to use the <join> which is
# essentially the opposite of <split>.


words = ["The","quick","brown","fox","jumps","over","the","lazy","dogs."]

space = " "
sentence = space.join(words)
print()
print(sentence)
```

```

sentence = ",".join(words)
print()
print(sentence)

sentence = "<~>".join(words)
print()
print(sentence)

sentence = "".join(words)
print()
print(sentence)

```

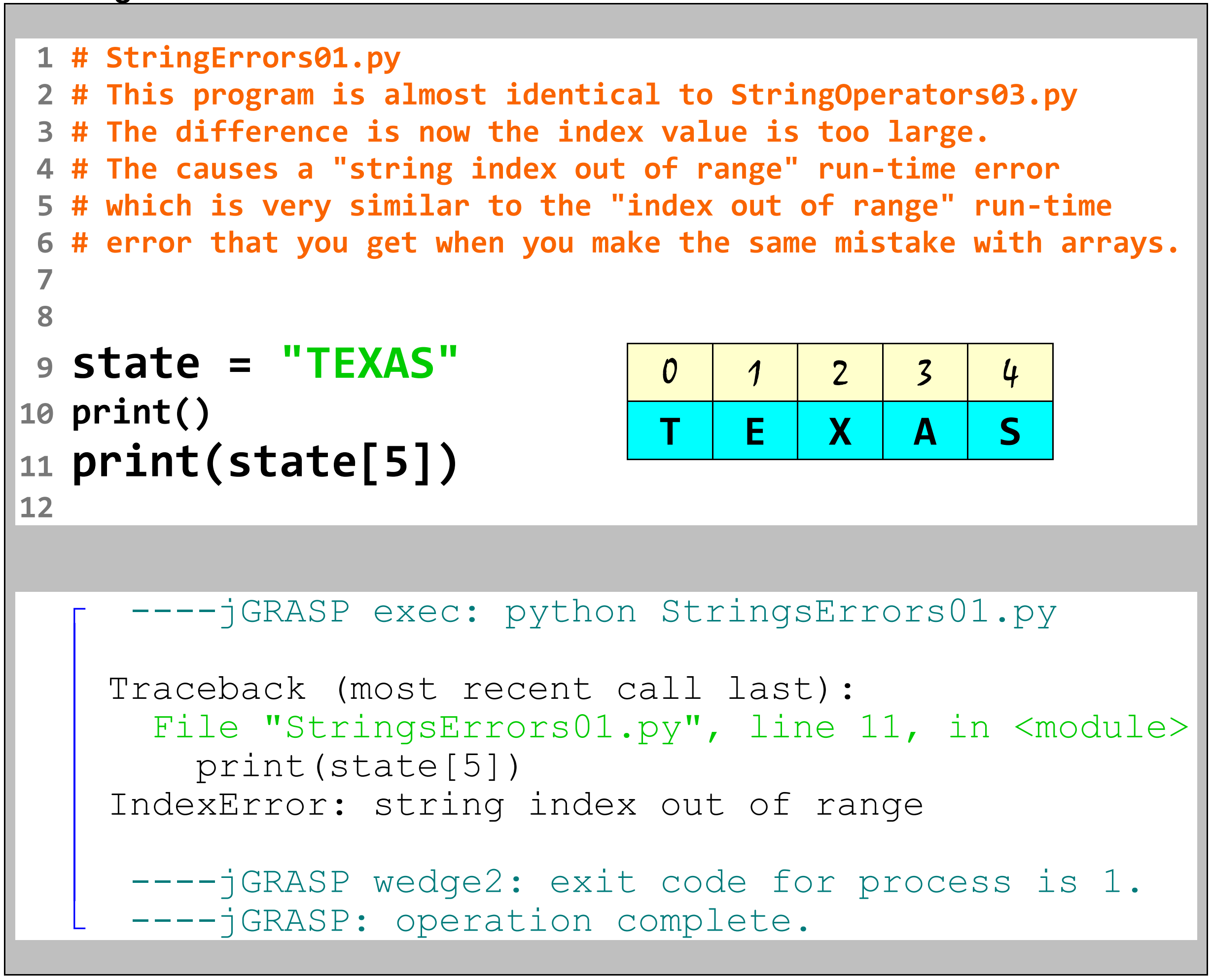
```
 ----jGRASP exec: python StringCommands23.py

The quick brown fox jumps over the lazy dogs.

The,quick,brown,fox,jumps,over,the,lazy,dogs.

The<~>quick<~>brown<~>fox<~>jumps<~>over<~>the<~>lazy<~>dogs.

Thequickbrownfoxjumpsoverthelazydogs.

 ----jGRASP: operation complete.
```

14.5 Common Errors with Strings

When working with string commands, as with working with any commands, various errors can occur if the rules are not followed. The final 6 programs in this chapter will show various *Compile*, *Run-time* and *Logic Errors* that can occur if string commands are not used properly.

Program **StringErrors01.py**, in Figure 14.38, causes an error that is similar to something you have seen before. With arrays, if you use an index that is too large (meaning bigger than the length of the array minus 1) you get an **Index Out Of Range** Error. We have already shown that in Python, a string is essentially an array of characters, and each character in the string has an index. Whether they are array indexes or string index, the same rule applies. The maximum index is the size of the string minus 1. If you go beyond that, you get a **String Index Out Of Range** Error.

Figure 14.38



```python
# StringErrors01.py
# This program is almost identical to StringOperators03.py
# The difference is now the index value is too large.
# The causes a "string index out of range" run-time error
# which is very similar to the "index out of range" run-time
# error that you get when you make the same mistake with arrays.


state = "TEXAS"
print()
print(state[5])

```

0	1	2	3	4
T	E	X	A	S

```
 ----jGRASP exec: python StringsErrors01.py

Traceback (most recent call last):
  File "StringsErrors01.py", line 11, in <module>
    print(state[5])
IndexError: string index out of range

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Errors can also occur when converting strings. Now, converting some to a string is no problem. Anything you type can be converted to a string. Problems can occur when you try to convert a string to something else.

Errors can also occur when converting strings. When converting other data types to strings, there is no problem. Anything you type can be converted to a string.

Problems can occur when you try to convert a string to something else like a number.

Program **StringErrors02.py**, in Figure 14.39, demonstrates the problem that can occur when converting a string to an integer. The **int** command requires that its string parameter contain an integer and nothing else. In this program, the string variable **n1** is storing the string literal **"3.14159"**, which contains a real number, not an integer.

Figure 14.39

```
# StringErrors02.py
# This program tries to use the <int> command
# to convert the string value in <n1> to an
# integer which is not possible.


n1 = "3.14159"
n2 = int(n1)

print()
print(n1)
print(n2)

```

```
 ----jGRASP exec: python StringsErrors02.py
Traceback (most recent call last):
  File "StringsErrors02.py", line 8, in <module>
    n2 = int(n1)
ValueError: invalid literal for int() with base 10: '3.14159'

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

You can also get an error when trying to convert a string to a real number with the **float** command. This is demonstrated by program **StringErrors03.py**, shown in Figure 14.40. The problem now is we are trying to convert the word **"Qwerty"** to a real number which is just not possible.

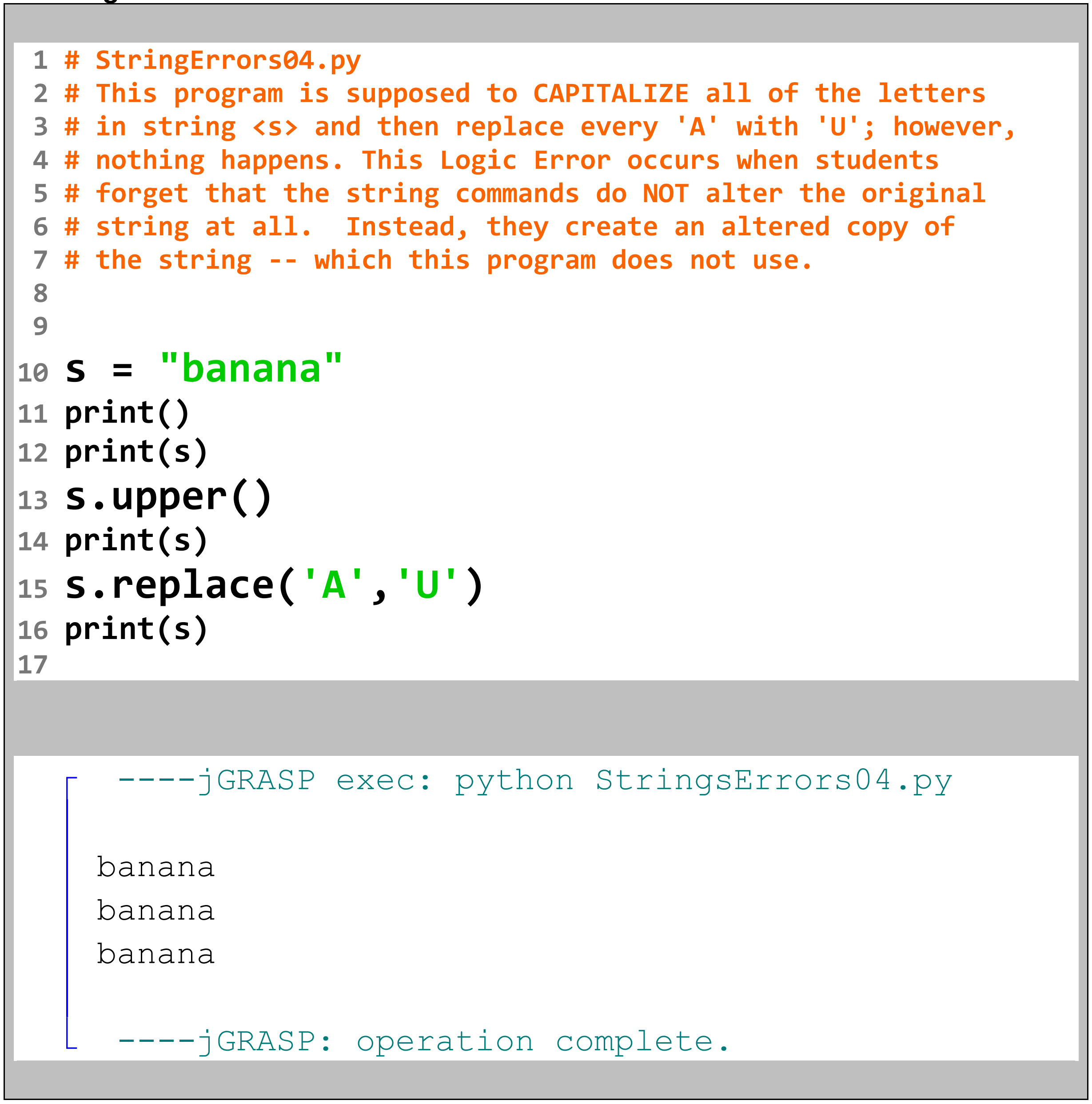
Figure 14.40

```
# StringErrors03.py
# This program is similar to the previous program.
# Now it tries to use the <float> command to convert
# the string value in <n1> to a real number which
# is also not possible.


n1 = "Qwerty"
n2 = float(n1)

print()
print(n1)
print(n2)

```

```
 ----jGRASP exec: python StringsErrors03.py
Traceback (most recent call last):
  File "StringsErrors03.py", line 8, in <module>
    n2 = float(n1)
ValueError: could not convert string to float: 'Qwerty'

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Program **StringErrors04.py**, in Figure 14.41, demonstrates the *Logc Error* that occurs when people forget that string commands do not actually alter the original string. In this program, string variable **s** stores string literal **"banana"**. The intention of the program is to CAPITALIZE all of the letters in string **s**, and then replace every occurrence of **'A'** with **'U'**. While this program does make use of the **upper** and **replace** functions, neither actually affects string **s**. Both merely return altered copies of the string, which this program does not use. The result is the program finishes with the exact same string with which it started.

Figure 14.41



```python
# StringErrors04.py
# This program is supposed to CAPITALIZE all of the letters
# in string <s> and then replace every 'A' with 'U'; however,
# nothing happens. This Logic Error occurs when students
# forget that the string commands do NOT alter the original
# string at all.  Instead, they create an altered copy of
# the string -- which this program does not use.


s = "banana"
print()
print(s)
s.upper()
print(s)
s.replace('A','U')
print(s)

```

```
  ----jGRASP exec: python StringsErrors04.py

 banana
 banana
 banana

  ----jGRASP: operation complete.
```

Program **StringErrors05.py**, in Figure 14.42, reviews how to fix the logic error of the previous program. The secret is to assign the return values from **upper** and

replace back to the original string **s**. See how this is done with the assignment operator (**==**) on lines 10 and 12. Now the program behave the way we want.

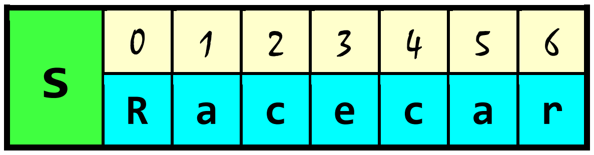
Figure 14.42

```
 1 # StringErrors05.py
 2 # This program demonstrates how to fix the logic error
 3 # of the previous program.  It also reviews what is
 4 # necessary to change the original <String> object.
 5
 6
 7 s = "banana"
 8 print()
 9 print(s)
10 s = s.upper()
11 print(s)
12 s = s.replace('A','U')
13 print(s)
```

```
  ----jGRASP exec: python StringsErrors05.py

 banana
 BANANA
 BUNUNU

  ----jGRASP: operation complete.
```

We will conclude this chapter with something that looks like it should be an error, but isn't. Program **StringErrors06.py**, in Figure 14.43, demonstrates what happens if your index values are too large when creating a *substring*. You may expect to get a **string index out of range** error, just like we did back in program **StringErrors01.py**, when we were trying to access a single character with an index that was too large.

To help explain this, let us look at string **s** below:



We see that string variable **s** stores string literal **"Racecar"**. By now you should now that anything like **s[10]** or even **s[7]** will crash the program with the a **string index out of range** error that I was just talking about. We also know that something like **s[1:4]** will return the substring **"ace"**, but what if I have **s[4:10]** or **s[10:]**? Then what would happen. Well, let us run the program and find out.

Figure 14.43

```
# StringErrors06.py
# This program demonstrates what happens when you create
# substrings with index values that are too large.
# Surprisingly, there is no error message.
# If the second number is too large, the substring will just
# go to the end of the string, just as if it were omitted.
# If the first number is too big, the substring is simply empty.


s = "Racecar"

print()
print(s[4:])
print(s[4:10])
print(s[10:])
```

```
 ----jGRASP exec: python StringsErrors06.py


car
car



 ----jGRASP: operation complete.
```

When we look at the output we see a couple of surprising results. First, when the second number is too large – as it is on line 14 – there is no error. The substring starts at the first index and goes through to the end of the string. This is exactly what happens when the second number is omitted – as it is on line 13. Note that both lines 13 and 14 produce the same output. As for what happens when the first number is too large – as it is on line 15 – it simply returns an empty string. That is why there is an extra blank line of output at the end.