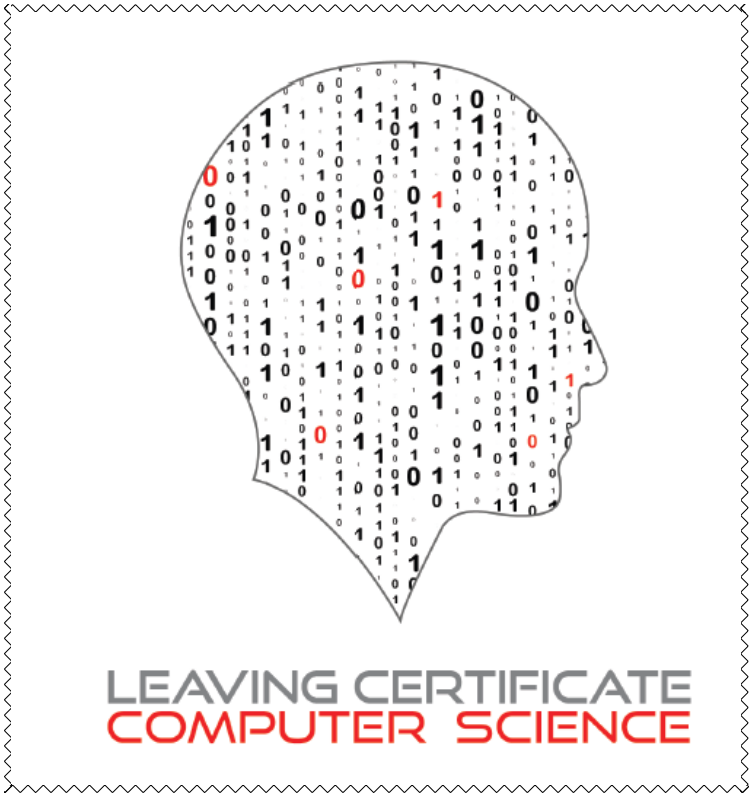
LC CS Python

Student Manual



Section 3

Strings

Name: _____

Introduction

In Section 1 we defined a string as any text enclosed inside quotation marks. Strings are important simply because, along with numbers, they are by far the most common type of data processed by computer programs and systems.

The value of a string can either be a *string literal* or any Python expression that results in a string. Some examples of simple string literals are listed below.

"Please enter your name: ”
"John Doe”
“+353-85-1234567”
“182 C 999”
“@PDSTcs Python CPD for #LCCS teachers”
“http://www.youtube.com/watch?v=hUkjib”

KEY POINT
The quotation marks are not part of the string

The individual symbols that make up a string are called *characters*.

Notice from the above examples that characters can be letters, numbers, spaces, punctuation marks and basically any symbol your keyboard will allow you to enter.

KEY POINT: A *string* is a sequence of characters enclosed by quotation marks.
Sometimes a string is referred to as an array (or list) of characters.

It is useful to think of the individual characters of a string being stored in consecutive locations of the computer’s memory. For example, string *Hello World* could be thought of as follows:

Internally, computers store unique numeric codes for each character and not the actual characters themselves.

When to use a string

It is relatively straightforward to understand what a string is. However, a more necessary skill lies in the ability to recognise the need to use strings (in computer programs).

The art of computer programming usually involves the representation of some real world phenomena by using a programming language such as Python. In other words, programmers use the features of Python to model real world phenomena.

The number of real world situations that can be modelled using computer systems is endless. Examples include buying, selling, order delivery, flight scheduling, medical systems, processing of examination results, news, entertainment and even socialising.

At an even finer level of detail, the string datatype is a feature of Python that is suitable for representing many real world things.

KEY POINT: A string should be used in a computer program to model any real world data that could be composed of any combination of letters, numbers and punctuation symbols.

Examples of string data include names, addresses, phone numbers, passwords, email texts, Facebook posts, SMS text messages, tweets, product codes, descriptions – the list is endless. In fact, most of the data you see on your mobile phone and on the world wide web are represented by strings.

As a beginner programmer, you should be aware of the fact that data which *looks numeric* may very often be represented in programs by strings. The reason for this is that these apparent numbers can often contain non-numeric data such as brackets, dashes or letters. Common examples include phone 'numbers', vehicle registration 'numbers' and Personal Public Service Numbers (PPSN). As a general rule, unless you need to do an arithmetic operation (add, subtract, etc) to your variable, you can declare it as a string.

String Operations

We already know we can display a string using the `print` command. But what are the other operations that can be carried out on strings?

By far the most important strings operations are *indexing* and *slicing*. Indexing is a programming technique used to access *individual characters* of a string. Slicing is a variation on this used to access *multiple continuous characters* from a string (known as a sub-string or a slice).

Other operations that can be carried out on strings are addition, multiplication, formatting and comparisons. Strings also support set operations such as in and not in. These set operations along with comparisons will be discussed in the section on programming logic.

Python also comes with a number of built in commands that can be used on strings as well as a comprehensive set of commands (known as *string methods*) specifically designed for working with strings.

String Indexing

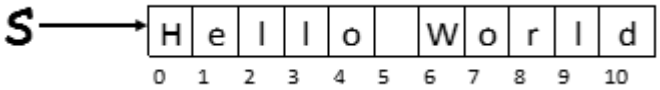
In order to understand string indexing (and slicing), it first necessary to understand the concept of an index. Consider the string `s` initialised in the line of code shown

```
s = "Hello World"
```

Internally, the computer represents the string `s` as a sequence of characters stored at contiguous memory locations. Each individual character is stored in its own separate memory location.

The individual characters in every Python string have a position. This position is relative to the first character in the string and is known as an *index*. Because the index is an offset from the first character, the index of the first character itself, in *every* Python, string is zero. Indices are said to be *zero-based*.

The diagram below depicts the index position of every character in the string `s`.



As can be seen the first character i.e. 'H' has an index of zero. This is important. The first character in a string always occurs at index position zero (this is often called the zeroth position), the second character at index position 1, the third character at index position 2 and so on. The string *Hello World* contains 11 characters and each character has a unique index ranging from 0 to 10.

In general, when there are n characters in a string the last character always occurs at index position $n - 1$. For example, if there are 5 characters in a string the last character occurs at index position 4.

Each individual character of a string can be accessed by enclosing the character's index inside square brackets The square brackets must appear directly after the name of the string. For example, the first element of the above string can be accessed using `s[0]`. So,

```
s[0]  → "H"
s[1]  → "e"
s[2]  → "l"
s[3]  → "l"
s[4]  → "o"
s[5]  → " "
s[6]  → "W"
s[7]  → "o"
s[8]  → "r"
s[9]  → "l"
s[10] → "d"
```

KEY POINTS

- ✓ Every character in a string has a unique index.
- ✓ The index is the character's position in the string.
- ✓ The first character is at index position zero.
- ✓ The last character in a string of length *n* is at index position *n-1*
- ✓ Indices are enclosed in square brackets

It is worth pointing out that Python, unlike many other programming languages does not support the concept of a character datatype. All of the single characters returned in the above example are actually strings i.e. single character strings of length 1.

Negative Indices

Python permits the use of negative numbers as indices.

The diagram to the right depicts the positive and negative indices of the string *Hello*.

As can be seen the last (rightmost) character of the string can be accessed using index -1. Working from right to left the index of each character is one less than its predecessor.

Programming Exercise

- In handout, do **Task 1** (shown below).

Experiment!

A **pangram** is a sentence that uses every letter of the alphabet at least once. Study the program below and see if you can predict what it does? Record your prediction on the right hand side.

Key in the program and make some changes? What happens?

```
1.  # Program to demonstrate basic string operations
2.
3.  # Initialise the string
4.  pangram = "The quick brown fox jumps over the lazy dog!"
5.  #          01234567890123456789012345678901234567890123
6.  # INDEXING
7.  print(pangram[0])
8.  print(pangram[1])
9.  print(pangram[2+4])
10. print(pangram[14])
11. print(pangram[8])
12. print(pangram[43])
13.
14. # The index can be any valid Python expression
15. pos = 17
16. print(pangram[pos])
17. print(pangram[pos+1])
18.
19. # A general pattern used to find the last character
20. print( pangram[len(pangram)-1] )
```

Prediction

What one question would I still like to ask in relation to this example?

Common Pitfalls

It is worth pointing out a couple of very common pitfalls relating to strings and index numbers

The first thing to note is that strings are *immutable*. This means that once a string has been initialised it cannot be changed. For this reason, it is not permitted to use the index operator ([]) on the left hand side of an assignment statement.

For example, the string *Cavan* could not be changed to *Navan* as follows.

```
town = "Cavan"
town[0] = "N"
```

This code results in a syntax error shown below:

```
Traceback (most recent call last):
  File "C:/PDST/Work in Progress/Python Workshop/src/Session 3
ts/Str Ops - syntax.py", line 3, in <module>
    town[0] = "N"
TypeError: 'str' object does not support item assignment
```

To 'change' the value of town to Navan the program needs to make a new assignment `town="Navan"` but be warned that the reference to the original string *Cavan* is now lost.

Secondly, if a program attempted to access a character in a string using an index that is too big (i.e. beyond the last character) for that string, Python returns a runtime error telling you that the index is *out of range*. For example, running the following code snippet would result in a runtime error being displayed because the index 5 is beyond the range of the string.

```
s = "Hello"
print(s[5])
```

```
Traceback (most recent call last):
  File "C:/PDST/Work in Progress/Python Workshop/src/Session 3
ts/Str Ops - syntax.py", line 2, in <module>
    print(s[5])
IndexError: string index out of range
```

Out of range errors (aka *out of bounds errors*) can be a source of great frustration even for more experienced programmers.

String Slicing

Thus far we have used indexing to access individual characters from a string. The technique of indexing can also be used to extract a sub-string from a string. This is known as slicing. The part of the string that is extracted is known as the slice (i.e. sub-string).

Slicing is useful when we want to extract a specific piece of information out of a longer piece of information. For example, we may want to extract a share price from a web page displaying stocks.

In order to extract a slice from a string we need to know the indices of the desired slice's start and end positions. The slice is taken by specifying these values inside the square brackets. The values are separated by a full colon.

The technique of slicing is demonstrated by the following short program.

```
1.  # Program to demonstrate basic string slicing
2.
3.  # Initialise the string
4.  pangram = "The quick brown fox jumps over the lazy dog!"
5.  #          0123456789012345678901234567890123456789012345
6.
7.  # Extract from index 1 up to but NOT including index 5
8.  print(pangram[1:5]) # "he q"
9.
10. # Extract from index 17 up to but NOT including index 19
11. print(pangram[17:19]) # ox
12.
13. print(pangram[:19])    # "The quick brown fox"
14. print(pangram[20:26]) # "jumps"
15. print(pangram[26:])    # "over the lazy dog!"
```

The colon delimits the start and end positions of the slice we are interested in extracting. The resulting slice runs from the starting index *up to, but not including* the end. If the start is missing it is taken to be zero i.e. the first character of the string. If end is missing it is taken as the index of the last character in the string.

Programming Exercise

- In handout, do **Task 2** (code shown above).
- In handout, do **Task 3.**

String Addition and Multiplication

Strings, just like numbers can be added and multiplied.

The operation of adding one string to another is commonly referred to as *string concatenation*. We say one string is concatenated to another string to form a new (and longer) string. The plus operator, '+' is used to concatenate two strings.

The programs below illustrate how the plus operator is used to concatenate strings.

PROGRAM 1

```
word1 = "Leaving"
word2 = "Certificate"
word3 = "Computer"
word4 = "Science"
subjectName = word1 + word2 + word3 + word4
print(subjectName)
```

PROGRAM 2

```
pangram = "The quick brown fox jumps over the lazy dog!"
#          01234567890123456789012345678901234567890123
print(pangram[:3] + pangram[16:])
```

PROGRAM 3

```
noun = input("Enter a singular noun: ")
print("The plural of "+noun+" is "+noun+"s")
```

Programming Exercise

- In handout, do **Task 4** (code shown above).

Log your thoughts.
Each of the above programs contains a deliberate subtle error.
Can you suggest any improvements?

String construction

String concatenation is a useful technique for building up a string made up of separate strings that come from different sources. Let's say we wanted to construct an output string based on the `pangram` string we used earlier.

```
"The quick brown fox jumps over the lazy dog!"
```

This time however we want to ask the user to enter the colour of the fox and the name of the animal to jump over. The output string will be constructed based on the values entered by the user. Example outputs could be:

```
"The quick red fox jumps over the lazy horse!"
"The quick orange fox jumps over the lazy hen!"
```

One solution is as shown here.

Observe how the `outStr` is built up bit by bit as the program progresses.

Can you see any similarities between this and the technique for keeping running totals described earlier?

```
# Initialise the output string
outStr = "The quick "

# Ask the user for a colour
colour = input("Please enter a colour: ")

# Concatenate the colour to the output
outStr = outStr + colour
outStr = outStr + " fox jumps over the lazy "

# Ask the user for an animal
animal = input("Please enter an animal: ")
outStr = outStr + animal
outStr = outStr + "!"

# Display the output
print( outStr )
```

String multiplication

Multiplication of strings in Python is not very common. When a string is multiplied by some integer n, a new string is produced. This new string is the original string repeated n times.

The technique of string multiplication in shown in the program below. The output generated is displayed to the right.

```
word1 = "Hello"
print(word1*3)
print(word1[1]*3)
print(word1[1:3]*3)
```

```
HelloHelloHello
eee
elelel
```

Programming Exercise

- In handout, do **Task 5** (shown below).

The string variable `alNum` is initialised as shown here.

```
alNum = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
#                1         2         3         4         5         6
#       0123456789012345678901234567890123456789012345678901234567890123
```

Match each index operation on `alNum` to the correct value in the middle

`alNum[3]`	ABCDEFGHIJKLMNOPQRSTUVWXYZ0
`alNum[52:62]`	d
`alNum[51:62]`	ABCDEFGHIJKLMNOPQRSTUVWXYZ
`alNum[26:53]`	A
`alNum[0:25]`	0123456789
`alNum[1]`	f
`alNum[5]`	b
`alNum[26:52]`	abcdefghijklmnopqrstuvwxyz
`alNum[26]`	Z0123456789

Strings Formatting

Sometimes it is useful to be able to display a string that is made up of several pieces of data without having to use the string construction techniques just described.

Formatting string expressions enable us to display more than one piece of data in a single `print` statement. Consider the following statement:

```
print("Hello %s" %"World")
```

The first string after the opening brackets on each line is called the *format string*. This is the string that Python displays once it has it formatted. In this example the format string is *Hello %s*

The %s is a placeholder for Python to insert a string value into the format string. %s is an example of a Python *formatting type code* (aka format specifier).

The % immediately after the format string tells Python that what follows is a *string formatting expression*. This expression contains the actual value(s) for Python to substitute into the format string.

In the above statement, Python replaces %s with the string literal, *World*, and the resulting formatted string *Hello World* is displayed.

Some common string formatting codes are listed below

- %s is used to format text (strings)
- %d is used to format integers
- %f is used to format floating point (decimal) numbers
- %.2f is used to format floating point numbers to 2 decimal places (rounded)

KEY POINT: The `print` command replaces each format code, *in strict sequence,* with a value of the appropriate datatype from the formatting expression.
If the format string contains more than one code, then the corresponding values in the expression must be separated by commas and enclosed by brackets.

In handout, do **Task 6** (shown below).

The four statements below each generate the same output. What output is displayed?

```
print("2 + %d = 4"    %2)
print("2 + %d = %d"   %(2, 4))
print("%d + %d = %d" %(2, 2, 4))
print("%d + %d = %d" %(2, 2, 2+2))
```

_____

_____

_____

The output generated by the following code is displayed to the right.

```
print("%s" %"String 1")
print("%s %s" %("String 1", "String 2"))
print("%s + %s = %d" %("2", "3", 2+3))
print("%d + %d" %(2, 3))
print("%d + %d = %d" %(2, 3, 2+3))
print("%f" %3.14)
```

```
String 1
String 1 String 2
2 + 3 = 5
2 + 3
2 + 3 = 5
3.140000
```

Programming Exercise

In handout, do **Task 7** (shown below).

Experiment!
See if you can figure out what the following code does.

```
print("%s" %3)
print("%d" %3.14)
print("%f" %3)
print("%f" %"Hi!")
```

Finally, it is worth noting that string formatting expressions can, and frequently do, contain variables and/or other Python expressions. This is exemplified in these two snippets.

```
msg = "Hi %s. How are you?"
name = "Hal"
print(msg%name)
```

```
import math
r = 5
print("Radius: %d, Area: %.2f" %(r, 2*math.pi*r))
```

In handout, do **Task 8** (code shown above).

Built in string commands

So, let's say we have a string s declared and initialised as follows:

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

Command	Description	Output
`len(s)`	Returns the length of the string, `s` This is the number of characters in the string.	44
`min(s)`	Returns the minimum item in `s`. See below.	`" "`
`max(s)`	Returns the maximum item in `s`. See below.	`"z"`

Another built in command that relates to strings is `str`. The `str` command is used to convert any object into string. This means a program can dynamically change the type of any object to a string type.

One practical use of `str` is to convert numbers to strings when we want to display them as part of an output message. Observe the way the output string is displayed by the following program.

```
1.  # Program to calculate a running total
2.
3.  # Initialise the variable
4.  runningTotal = 0
5.
6.  # Keep a running total of the amounts entered
7.  price1 = int(input("Enter the 1st price: "))
8.  runningTotal = runningTotal + price1
9.  price2 = int(input("Enter the 2nd price: "))
10. runningTotal = runningTotal + price2
11. price3 = int(input("Enter the 3rd price: "))
12. runningTotal = runningTotal + price3
13.
14. # Display the output
15. print("Total amount is €%s" %str(runningTotal))
```

KEY POINT: Python is *dynamically typed*. This means that when a program is running, numeric data can be converted to strings and vice versa.

Coding Systems (`ord` and `chr`)

Two other built in Python commands that relate to strings are **`ord` and `chr`**

Programmers should be aware that all data is represented internally by computers using a coding system. A *coding system* is one which uses combinations of binary digits to represent data values uniquely. **ASCII** (American Standard Code for Information Interchange) and **Unicode** are the names of two very widely used coding systems. An illustration of the full ASCII character set is shown below.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

The two built-in functions – `ord` and `chr` - are used to convert back and forth between characters and ASCII.

`ord(c)` Returns the ASCII representation (or Unicode code point) of the character `c`.

`chr(i)` This is the inverse of `ord`. The function returns a single character as a string whose ASCII representation (or Unicode code point) is the integer .

In handout, do **Task 9** (shown below).

The programs below illustrate the use of `ord`.

```
1. print(ord('A'))
2. print(ord('A')+25)
3. print(ord('Z'))
4. print(ord('a'))
5. print(ord('1'))
```

Line 1 outputs 65. This is the ASCII representation for the character 'A'

Line 2 outputs 90, the ASCII representation for the character 'Z' (25 letters from 'A'

Use the ASCII table shown on the previous page to predict the outputs of lines 3, 4 and 5 of the above program.

The programs below illustrate the use of `chr`.

```
1. print(chr(65))
2. print(chr(90))
3. print(chr(97))
4. print(chr(49))
```

Line 1 outputs character 'A'
Line 2 outputs character 'Z'
Line 3 outputs character 'a'
Line 4 outputs character '1'

Experiment!
Try the following programs and see if you can explain what is going on.

```
print(chr(ord('A')))
print(ord(chr(64)))
```

```
inStr = input("Enter any character: ")
outStr = chr(ord(inStr)+1)
print(outStr)
```

In handout, do **Task 10** (shown below).

Programming Exercise

A Caesar cipher is a way of encoding strings by shifting each letter by a fixed number of positions (called a key) in the alphabet. For example, if the key value is 1, the string 'LCCS'. would be encoded as 'MDDT'. Each letter of LCCS is moved on by one.

The short program below prompts a user to enter a single character and then it calculated and displays the character with the next ordinal number e.g. if the user enters A the program will display B.

```
inStr = input("Enter any character: ")
outStr = chr(ord(inStr)+1)
print(outStr)
```

Type in the program and test it. Once you understand what the program does change it so that it "encodes" a 6 letter string using a key value of 1 e.g. "Python" becomes "Qzuipo".

In order to complete this task, you will need to understand:

- ✓ Variables and assignments
- ✓ Strings (indexing and concatenation)
- ✓ Data representation (ASCII/Unicode)
- ✓ How to use the `ord`, `chr` and `print` built-in functions

String Methods

String methods are special commands that can be used to manipulate and perform common/useful operations on strings.

The official Python documentation https://docs.python.org/3/library/stdtypes.html#string-methods lists and describes over 40 built in string methods.

The program below demonstrates five of these – `capitalize`, `upper`, `isupper`, `islower` and `find`.

```
1.  # Program to demonstrate the use of common string methods
2.
3.  pangram = "the quick brown fox jumps over the lazy dog!"
4.
5.  print(pangram.capitalize())
6.  print(pangram.upper())
7.  print(pangram.isupper())
8.  print(pangram.islower())
9.  print(pangram.count("o"))
10. print(pangram.find("fox"))
11. print(pangram.find("Fox"))
12. print(pangram.find("the"))
13. print(pangram.find("z"))
```

When the above program is run it generates the following output.

```
The quick brown fox jumps over the lazy dog!
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG!
False
True
4
16
-1
0
37
```

Each of the nine lines of output correspond, in sequence, to the result of the string methods used inside each of the nine `print` statements. The program, along with the output it generates should be studied carefully again after reading the method descriptions on the next page.

As can be seen, the string methods operate using the same dot notation as we used for `turtle` and `pygame` objects earlier. For any string variable `s`, we can use a string method by typing `s`, followed by a dot, followed by the method's name. This is denoted as follows:

<string-variable-name>.<method-name>

Method Name	Description
`s.capitalize()`	Returns a new string with the first letter of `s` in capital letters
`s.upper()`	Returns a new string with all the letters of `s` in upper case
`s.isupper()`	Returns the value `True` if all the letters in `s` are in upper case. If all the letters are not in upper case the method returns `False`. `True` and `False` are both Python keywords will be explained in the next section on programming logic.
`s.islower()`	Returns the value `True` if all the letters in `s` are in lower case. If all the letters are not in lower case the method returns `False`. `True` and `False` are both Python keywords will be explained in the next section on programming logic.
`s.count("o")`	This methods counts the number of times any string e.g. *o* occurs in the string `s` and returns the answer.
`s.find("Fox")`	This method looks for a string e.g. *Fox* in the string `s` and if it finds it returns the index position of the F. If the search sting is not found the method returns `-1`.
`s.replace(t, u)`	Replaces all occurrences of `t` in `s` with `u`. `s`, `t` and `u` are all strings. `s` remains unchanged if it does not contain `t`.

In handout, do **Task 11** (shown below).

The code below initialises four string variables

```
# Initialise four string variables
lowerLetters = "abcdefghijklmnopqrstuvwxyz"
upperLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
allLetters = lowerLetters+upperLetters
digits = "0123456789"
```

Match each Python statement to the correct value shown on the right

`print(lowerLetters.upper())`	a
`print(upperLetters.lower())`	Z
`print(upperLetters.find("h"))`	abcdefghijklmnopqrstuvwxyz
`print(lowerLetters.find("h"))`	0123456789
`print(digits.count("123"))`	ABCDEFGHIJKLMNOPQRSTUVWXYZ
`print(lowerLetters.capitalize())`	-1
`print(digits.count("0"))`	1
`print(digits.lower())`	z
`print( min(allLetters) )`	A
`print( max(allLetters) )`	7
`print(allLetters.count("a"))`	0
`print(allLetters.count("aA"))`	abcdefghijklmnopqrstuvwABCABCDEFGHIJKLMNOPQRSTUVWXYZ
`print(allLetters.count("ABC"))`	Abcdefghijklmnopqrstuvwxyz
`print(digits.replace("0", "1"))`	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
`print(allLetters.replace("ABC", "ABC"))`	6
`print(allLetters.replace("xyz", "ABC"))`	1123456789

Additional Notes (Sequences)

Strings like almost everything else in Python are examples of objects. The string type is a core data type that is built into the language. Some other examples of built in types include numbers, lists, dictionaries and files. The official documentation on all Python's types can be viewed online at https://docs.python.org/3/library/stdtypes.html and is well worth a visit.

More specifically Python classifies strings as part of a more fundamental kind of object known as a sequence. A sequence is defined as an ordered collection of objects

Python supports three basic sequence types (`list`, `tuple` and `range`), a text sequence type (`str`) and a number of binary sequence types use to work with binary data.

All sequence types share a common set of operations, called sequence operations. The table below, which is taken directly from the official Python documentation, lists these common sequence operations in ascending order of precedence.

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Note that concatenation, multiplication, indexing, and slicing – all described in this section in the context of strings – are all common sequence operations and, therefore, are equally applicable to all the other sequence types. Sequences are also distinguished from one another by their own specific set of operations that are not available to other types. For example, the string methods described earlier in this section are specific to string sequences and cannot be used on other sequence types. Finally, it is worth noting that unlike many other high level programming languages, Python does not provide any built-in support for the character datatype. In Python, a single character is simply a string of length one.

KEY POINT: Strings are sequences of one character strings.