Name: Date:

Lesson 3: The Caesar Shuffle

ord() and chr()

At the end of Lesson 2 we met a built-in Python function called `ord()` which takes a single parameter character and returns a special integer value. We also mentioned built-in partner function called `chr()` which takes that special integer value and returns the original character. Here is the full code for last week's Code Quest:

```
def encodeString(s):
    r=[]
    for c in s:
        r.append(ord(c))
    return r

def decodeList(ls):
    s=''
    for i in ls:
        s=s+chr(i)
    return s

lst=encodeString("Secret string!")
print(lst)
st=decodeList(lst)
print(st)
```

TRY IT OUT #1: Try copying this code as above into IDLE in edit mode. Make sure you understand how it works and that you are able to run it. Note that from now on, everything we try out should be done in edit mode and not in the interpreter so you can easily fix bugs in your code.

The code you copied will result in the following output when you run it:

```
[83, 101, 99, 114, 101, 116, 32, 115, 116, 114, 105, 110, 103, 33]
Secret string!
```

The special integer values that `ord()` and `chr()` translate between are called **ASCII** values.

ASCII

ASCII stands for the American Standard Code for Information Interchange. It is a special scheme for **encoding characters** that uses **128 values** from **0** to **127** as outlined in the encoding table below. ASCII dates back to the 1960's and is how **text** was originally processed by computers. Most modern character-encoding schemes to this day are built upon ASCII though they support many additional characters. Note that in ASCII the numbers '0' to '9' are in a continuous range from 48 to 57, the letters 'a' to 'z' in a continuous range from 97 to 122 and 'A' to 'Z' in a continuous range from 65 to 90. We can use this fact to help us build the Caesar cipher later on.

In Python, you use the two built-in functions you met called **`ord()`** and **`chr()`** to convert between a character letter and its underlying ASCII representation as follows:

```
>>> ord('a')
97
>>> ord('b')
98
>>> chr(97)
'a'
```

The following ASCII table shows both ASCII and character values. ASCII values are in the "Dec" column with corresponding characters in the "Chr" column. `ord()` converts from "Chr" to "Dec" value and `chr()` converts from "Dec" to "Chr". You can ignore the Hx, Oct and Html columns.

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

TRY IT OUT #2: Try converting some characters to ASCII and back again using `ord()` and `chr()` and make sure you check that the ASCII values agree with those in the encoding table.

Working with alphabetic characters

Since ASCII values are just integers, it is possible to add to or subtract from them. This allows you to move a character when you translate back. For instance, we can shift character `'a'` forward by adding 3 to its ASCII value which after translating back results in `'d'`:

```
>>> chr(ord('a')+3)
'd'
```

There are a few important built-in string functions it is important to know about:

- `s.isalpha()`: This function checks whether the string is alphanumeric. That means the string contains ONLY characters in the range '0'-'9' OR 'a'-'z' OR 'A'-'Z'.
- `s.islower()`: This function checks whether the string is alphanumeric. That means the string contains ONLY characters in the range '0'-'9' OR 'a'-'z' OR 'A'-'Z'.
- `s.lower()`: This function returns a NEW lowercase version of the string
- `s.upper()`: This function returns a NEW uppercase version of the string

Note that `isalpha()` and `islower()` both return a **boolean** variable of type **bool** which can either be **True** or **False**. Here are some examples of how to use these functions:

```
s="Hello World"
print(s)
print(s.isalpha())
print(s.islower())
s="HelloWorld"
print(s)
print(s.isalpha())
print(s.islower())
s=s.lower()
print(s)
print(s.isalpha())
print(s.islower())
```

You should get the following output which reveals that the string "`helloworld`" holds True for both `isalpha()` and `islower()`:

```
Hello World
False
```

```
False
HelloWorld
True
False
helloworld
True
True
```

TRY IT OUT #3: Try out built-in string functions `isalpha()` and `islower()` on a number of strings of your own choosing to make sure you understand how they work. Also make sure you understand how `lower()` and `upper()` work.

Looping round an alphabet

At this point, we know how to use `lower()` to lowercase a string and then use `ord()` to generate ASCII values for each `isalpha()` alphanumeric characters in that string. We could use this to implement a **shift cipher** where our input string is **scrambled** by an **offset** value. For instance the string "`hello`" **right shifted** by 2 becomes "`jgnnq`". With any right shift, we need to address the characters at the end of the lowercase alphabet. If we simply shifted them right too, they would go outside the 'a'-'z' range and would no longer be alphanumeric. We need to shift them BACK round the range to the front part of the lowercase alphabet so '`xyz`' -> '`zab`'. To do this we need to **subtract** 26 from any character over the `ord('z')` limit of the lowercase ASCII alphabet:

```
def shiftCharacter(c,offset):
    if c.isalpha():
        n=ord(c)
        n=n+offset
        if n > ord('z'):
            n=n-26
        elif n < ord('a'):
            n=n+26
        translated=chr(n)
    else:
        translated=chr(c)
    return translated

s=''
offset=2
for c in 'abcdefghijklmnopqrstuvwxyz':
    s=s+shiftCharacter(c,offset)
print(s)
```

TRY IT OUT #4: Try copying the above `shiftCharacter()` function plus test code. Make sure you understand how it is working. Try it out with different positive and negative offset values corresponding to right and left shifts.

Here is the output you should get where every character is offset by +2 and we loop round to '`ab`' for '`yz`' at the end:

```
cdefghijklmnopqrstuvwxyzab
```

The Caesar Cipher

If we use an offset of +3 in the `shiftCharacter()` function above, we have the basic building block for the **Caesar cipher**. This is one of the simplest and most widely known encryption techniques. Here is a visualisation of a right shift encoding by +3 followed by a left shift back:

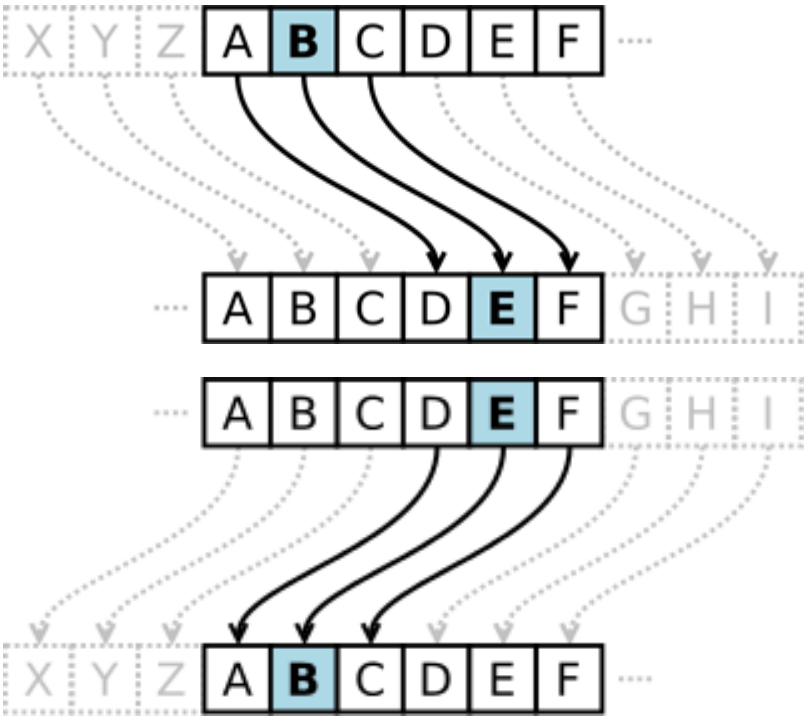

Diagram 1: Right and Left shift Caesar cipher example, offset=3

This offset of 3 was used by the Roman ruler Julius Caesar (100 B.C. – 44 B.C.) 2000 years ago for his secret communication in Latin. He substituted each letter of the alphabet with a letter three positions further along. Later, any cipher that used this "displacement" concept for the creation of a cipher alphabet, was referred to as a Caesar cipher and variations on the same method have been used down the ages ever since right up to the present day. Most of those using the Caesar cipher

preceded ASCII and the computer age! Instead they used cipher disks to shift along a fixed alphabet by a set amount. See the picture below for an example of a cipher disk. We can use Python to write a programming cipher disk which shifts along through the letters within the alphabet range '`a`' to '`z`'.

Diagram 2: Cipher disk, offset=2

Before we look at how to do this programmatically, let's try and make our own cipher disk from card!

TRY IT OUT #5: Make a cipher disk! You will need to get hold of two circular cardboard disks connected with a paper fastener. There are some instructions plus disks you can print and cut out here: http://www.nsa.gov/change/_files/solution/cipher_disk.pdf

Bringing it all together ….

We can now bring together all the building blocks you've learnt about this week to build our own version of the Caesar cipher in Python for our next Secret Code Quest!

SECRET CODE QUEST!

Write a function called `encodeCaesar()` that takes TWO parameters – a string to be encoded plus an integer offset. The function should return a shifted output as with the Caesar cipher described above. You can use a combination of all the techniques we've discovered in this Lesson to develop your answer. In particular, you will need to lowercase the string and then call `shiftCharacter()` on each character in the string. Your output will be a string built-up one character at a time. Here is some example usage and output:

```
s="This is a secret message!"
offset=3
print(encodeCaesar(s,offset))


qklv lv d vhfuhw phvvdjh!
```

Once you have this working, you could write out your encoded messages on paper and use Python to help you "decode" them back to their unreversed form. To do that you will need to write the opposite `decodeCaesar()` which takes the encoded string plus the original offset and returns the original unencoded string. Remember you will need to ensure the original offset is reversed by being made the **negative** opposite of what it originally was.