CSE231 - Lab 03

Strings, `enumerate()`, `in`

What are strings, really?

We've seen strings being used quite a bit now. It's the fundamental way we have to represent words or sentences in our programs.

In order to accomplish this, Python is storing the ***order*** of characters we put into our string. This order of characters is how we, as humans, read the text we see on screen.

In order to store all of these characters in a specific way, Python is putting them into a ***container***, or sometimes called a ***collection-type***. That's really all strings are -- a collection of ASCII characters in a specific order. A lot of the concepts we learn about strings here, are concepts that carry over to a lot of the other container types we'll be learning about.

Things About Strings / `len()`

Let's create a string and use some functions on it to see how they interact. The `len()` function simply gives the number of elements in any collection of items, in this case, we have the string "`Hello World`". There are 11 characters. As you might have noticed, the space in between the two words ***is*** counted as a character.

```
my_str = "Hello World"

print(my_str)    # Hello World

print( type(my_str) )    # <class 'str'>

print( len(my_str) )    # 11
```

Indices

Since strings are a collection, they can be “indexed” into. We can extract a single element from it, (in this case, an ASCII character), and use it for our own purposes. ***Indices always start from 0!***

```
my_str = “Hello World”

my_char = my_str[1]    # my_char now holds the string “e”
```

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

(Negative) Indices

Something unique to Python is the concept of “negatively-indexing” a container-type. Really comes in handy when trying to access the back-half of a container.

```
my_str = “Hello World”

my_char = my_str[-2]  # my_char now holds the string “l”
```

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

Substrings / Slicing

We can extract *subsets* of the string by “slicing”. The colon operator is saying something to the effect of: “extract all indices from here to here".

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

```
print( my_str[6:10] )    # ‘Worl’

print( my_str[6:] )    # ‘World’

print( my_str[:5] )    # ‘Hello’
```

Substrings / Slicing (cont.)

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

```
print( my_str[0:5] )    # ‘Hello’

print( my_str[3:-2] )    # ‘lo Wor’

print( my_str[0:90] )    # ‘Hello World’
```

Extended Slicing

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

```
print( my_str[0:11:2] )  # HloWrd ; same as my_str[::2]

print( my_str[::-2] )  # drWolH ; same as my_str[-1::-2]

print( my_str[1::2] )  # el ol
```

Strings with the Addition Operator

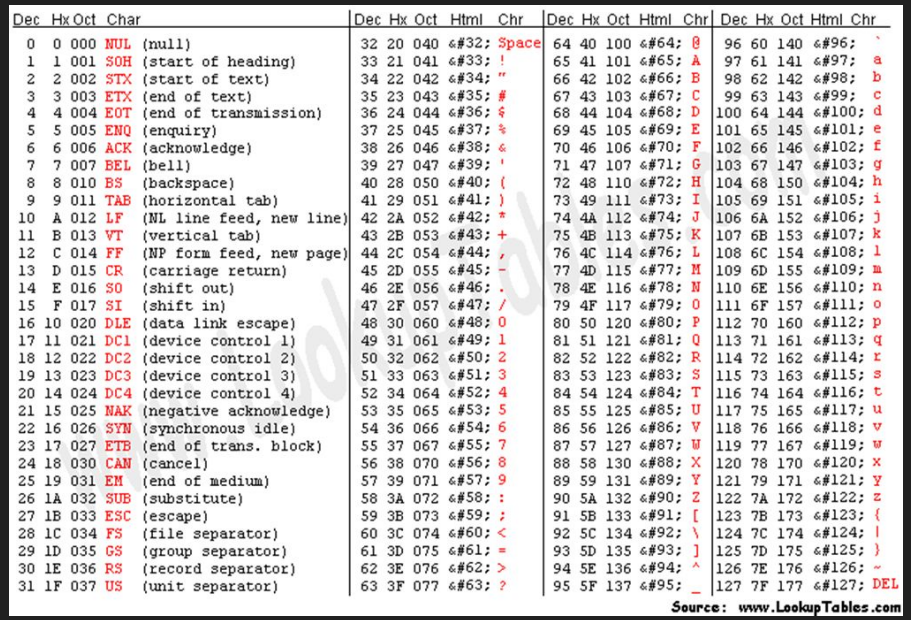
We've used ints and floats with the addition operator a lot now, but weirdly enough, strings *also* have special compatability with the addition operator.

We can build a new string by adding characters or entirely different strings.

```
my_str = ""

my_str += "Hello World."  # my_str is now "Hello World."

my_str += "    "  # my_str is now "Hello World.    "

my_str += "a"  # my_str is now "Hello World.    a"
```

Logical String Operations

You might've done this on accident in the past. Despite thinking you might not be able to, you can in fact, use comparison operators on strings.



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

Logical String Operations (cont.)

For a majority of operations, you're likely going to be dealing with the uppercase and lowercase versions of the ASCII alphabet.

You can think about letter comparisons like this; there is an all-uppercase version, and an all-lowercase version of the alphabet stored in your computer. The all-uppercase version comes before the all-lowercase version, where each letter in both versions are numbered sequentially.

So imagine listing them all from 1-52, starting from uppercase "A", and going all the way to lowercase "z".

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

Logical String Operations (cont.)

```
print(“A” < “B”)    # True, “A” comes before “B” in the uppercase alphabet

print(“A” < “b”)     # True, since “b” comes much later than “A”, in the lowercase alphabet

print(“a” < “B”)     # False, since “a” comes after every letter in the uppercase alphabet

print(“a” < “b”)     # True
```

When we’re checking strings beyond a single character, then the comparison goes letter by letter.

```
print(“hi” < “hello”)    # False, since “i” comes after “e” in the lowercase alphabet

print(“hI” < “hello”)    # True, since “I” comes before “e” in the uppercase alphabet
```

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

Method Functions (Methods)

Before we talk about string-specific method functions, what are method functions in general?

A method function is a type of function that only interacts with *a certain type*, such as an `int`, `float`, `bool` or `str`. We normally invoke non-method functions (such as `print()`) like this:

```
function_name( {argument}, {argument}, ... )
```

But a method function is different, where instead we call it ***on*** the object using a '`.`'.

```
{object}.method_name( {argument}, {argument}, ... )
```

String Methods

All of these method functions return a boolean `True` or `False`, they do not take any arguments.

```
# .isupper() - Checks if entire string is uppercase

# .islower() - Checks if entire string is lowercase

# .isdigit() - Checks if entire string is numeric

# .isalpha() - Checks if entire string is all letters

my_str = “hello world”

print(my_str.islower())    # True
```

String Methods (cont.)

These method functions return *copies* of the original string but modified by their functionality.

```
# .lower() - Lowercases all uppercase characters

# .upper() - Uppercases all lowercase characters

my_str = “Hello World”

my_str = my_str.upper()    # my_str is now “HELLO WORLD”

my_str = my_str.lower()    # my_str is now “hello world”

# stating ‘my_str.upper()’ (or .lower()) by itself does not modify

# the original variable, my_str
```

`.find()`

`.find(sub[, start[, end]] )`

`sub` - a substring you're attempting to find, type:str

`start` (optional) - starting index for the substring search, type:int

`end` (optional) - ending index for the substring search, type:int

If `sub` exists, `.find()` returns the index of ***first*** occurence. If it doesn't exist, it returns -1.

`.rfind()`

`.rfind(sub[, start[, end]] )`

`sub` - a substring you're attempting to find, type:str

`start` (optional) - starting index for the substring search, type:int

`end` (optional) - ending index for the substring search, type:int

If `sub` exists, `.rfind()` returns the index of ***last*** occurence. If it doesn't exist, it returns -1.

`.replace()`

`.replace(old, new [, count])`

`old` - old substring you want to replace

`new` - new substring which would replace `old`

`count` (optional) - the maximum number of times you want to replace instances of `old` (if there are multiple instances)

If `count` is not specified, `.replace()` will substitute *all* occurrences of `old` with `new`. It returns a *copy* of the string with the replaced substrings.

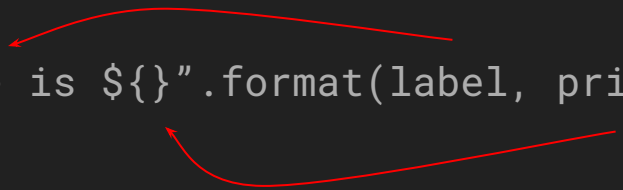
L3-1

.format()

`.format()` is a special method function that can replace parts of your string with whatever parameters you feed it. It offers a lot of customizability with how the string is presented. I'm going to cover a lot with this because Enbody *loves* putting `.format()` questions on his exams.

One super convenient use is to just replace parts of your string with whatever object/type you need it to be. You define the spaces that need to be filled with curly braces inside the string.

```
price = 2000

label = "Price"

print("{} is ${}".format(label, price))    # "Price is $2000"
```



`.format()` (cont.)

The curly braces can do a lot more, however. They can take special, character-based arguments that can adjust the way the incoming value is presented in the string.

```
{:[align] [width] [,] [.precision] [type] }
```

`align` - “<”, “>”, “^”, the direction of alignment (think of it as an arrow)

`width` - integer value, the amount of space allotted for the incoming value

`precision` - integer value, the variable decimal place to round to

`type` - ‘s’ (string-style), ‘d’ (int-style), ‘f’ (float-style), the presentation of the value

A .format() example

```
price = 2000

print("{:>10,d}".format(price))
```

This prints " 2,000", the string allocates 10 spaces (`width`), with value "`2000`" taking up the right-most 5 spaces since we used the right-justify, ">", argument (`align`).

```
{:[align] [width] [,] [.precision] [type] }
```

We also used the ', ' argument, which adds thousands-separator commas to our number where appropriate.

Another, more complicated `.format()` example

```
price = 2459.4678

print("{:^20,.2f}".format(price))
```

We end up getting " 2,459.47 ". price gets rounded to the 2nd decimal place (denoted by the ".2"), we used the "^" character for our alignment, (meaning we center it), and we center it within 20 spaces.

The alignment characters are "<", "^", and ">". Think of the character as an arrow that denotes which way the alignment goes towards. You probably won't be using "<" much since everything is left-justified by default. Center ("^") and right (">") tend to be the more useful ones. There is another that we're not going to discuss.

One last note on `.format()`

If you are using center justification, “^”, in an even amount of width and an odd amount of space taken up by the value (or vice versa), Python will always prefer the left-most side. Example:

```
price = 2

print(“{:^4}”.format(price))
```

Python will print “ 2 “, note the uneven amount of space around the ‘2’, since we’re trying to center-justify a single space within 4 empty spaces.

https://docs.python.org/2/library/string.html#format-string-syntax

`enumerate()`

In my opinion, this is probably the best function in any programming language, ever.

`enumerate()` is much like how you use `range()` in for-loops. Where `range()` will spit out the index, `enumerate()` will spit both the value it's iterating through, ***and*** the index.

We'll go through an example, since it's hard to describe what this function does theoretically.

`in`

I said on the first day that there were two other keywords that return booleans that we wouldn't be discussing until later. This is one of those two.

'`in`' comes in handy when checking if an element exists inside a collection, and as we talked about at the beginning of this lecture, a string *is* a collection. We can check if certain substrings are *in* our string without having to use something like `.find()`.

```
my_str = “Hello World”

print(“Hello” in my_str)  # True

print(“lo “ in my_str)  # True

print(“e” in my_str)  # True
```