Lecture 11

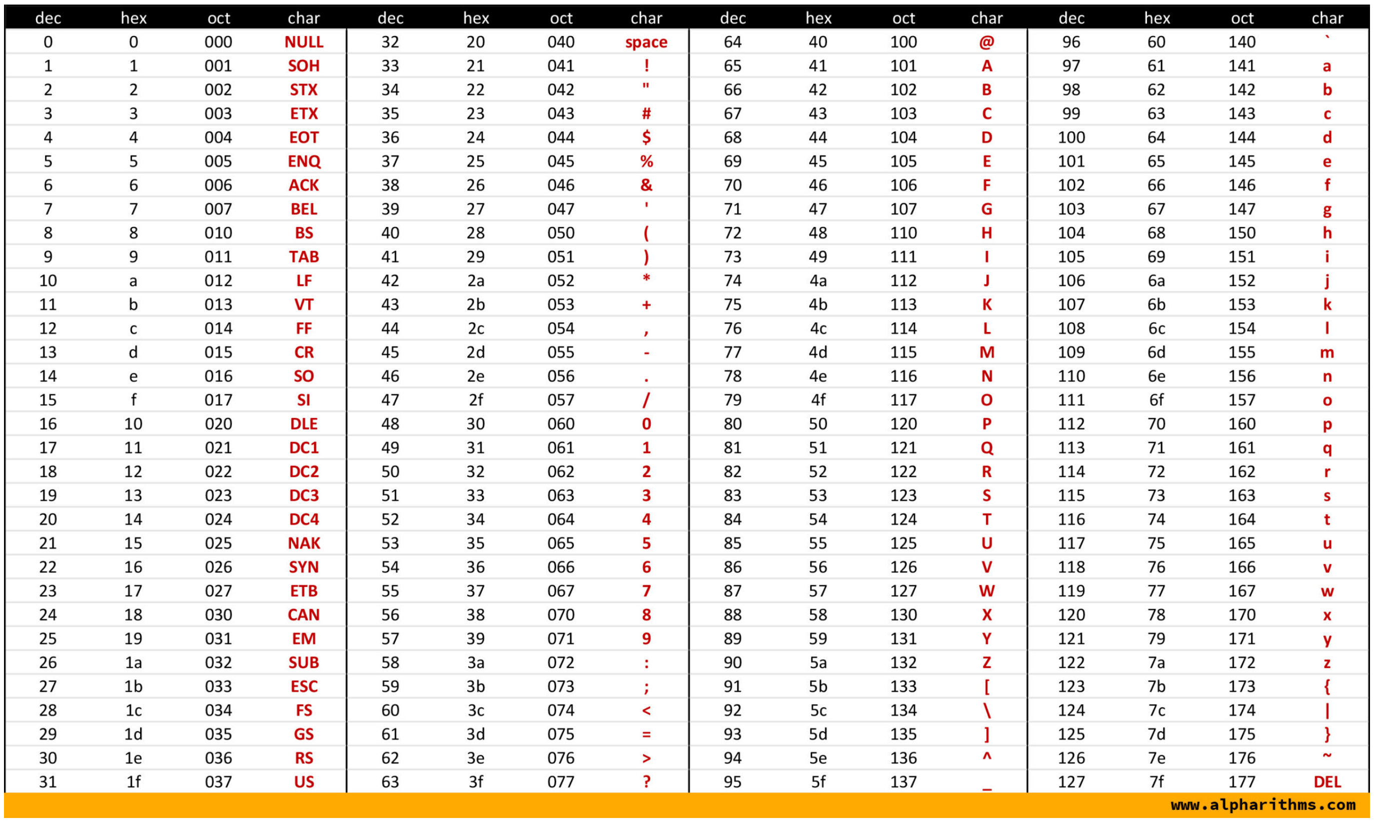
Manipulating Strings

12 Ventôse, Year CCXXX

Song of the day: ***Heart Attack*** *by LOONA/Chuu (2017).*

Part 0: *Character Representation (Continued)*

Recall our conversation on the ASCII table:



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

www.alpharithms.com

Figure 1: ASCII Table (**Source**).

ASCII stands for *American Standard Code for Information Interchange*, and is a means of encoding characters for digital communications. It was originally developed in the early 1960s as early networked communications were being developed.

Strangely, this table was developed to encode characters in 7-bits, so it contains a maximum of $(1111111)_2$, or $(127)_{10}$, characters.

Now, as a good number of us are not from Anglophone countries, this table will seem restrictive —and it is. Romance and other Germanic languages contain a good number of accent marks, for instance (etc. `señal` , `coração` , `straße`). Even worse, more than half of the world population doesn't even use a Latin-based alphabet in daily communication—the **cyrillic** and **Greek** alphabets, syllabaries like the ingenious Korean **hangul**, let alone the topographic **hànzì, kanji, and hanja** of China, Japan, and Korea, are completely ignored.

Now, of course, we have what is known as **unicode**, which can encode characters in 8-, 16-, and even 32-bits to accommodate to these important sets of characters.

So how do we use this information in Python—and what is it good for?

To find out a character's ASCII equivalent, we use the `ord()` , or **ordinal**, function:

```
>>> ord("a")
97

>>> ord("A")
65

>>> ord("ñ")
241

>>> ord("疲")
30130

>>> ord("δ")
948
```

And, conversely, to find the corresponding character to any given positive integer, we use `chr()` :

```
>>> chr(123)
'{'

>>> chr(42)
'*'

>>> chr(512)
'Ȁ'
```

Part 1: *String Traversal (Continued)*

Let's continue our conversation on strings with a quick program. Write a program that asks the user for two strings `first_string` and `second_string`. Then, your program will print all the characters from the first string that appear in the second string. Write three versions of this program:

1. Using `for`-loop string sequencing.
2. Using `for`-loop string indexing.
3. Using string indexing, but with a `while`-loop.

Here's a sample output of how your program could behave:

```
Enter a string: Rickenbacker 4000C Bass Guitar
Enter a second string: Rickenbacker 330 Electric Guitar
R
i
c
k
e
n
b
a
c
k
e
r
 
0
 
e
c
t
r
i
c
 
G
u
i
t
a
r
```

There are two ways we could approach this. The quickest and perhaps more intuitive one is use the second string as a sequence, and then check for membership of each of its letters in the first string. We do this by using `in`, the membership operator:

```python
first_string = input("Enter a string: ")
second_string = input("Enter a second string: ")

for letter in second_string:
    if letter in first_string:
        print(letter)
```

Code Block 1: Checking for membership by treating `second_string` as a sequence.

If we didn't want to use `second_sentence` as a sequence, and instead wanted to use indices, our `for`-loop would look something like this instead:

```python
first_string = input("Enter a string: ")
second_string = input("Enter a second string: ")

second_string_length = len(second_string)

for index in range(second_string_length):
    letter = second_string[index]
    if letter in first_string:
        print(letter)
```

Code Block 2: Checking for membership by using indices. Note the use of `len()`.

Both ways work and are actually equally efficient. Which one to use depends largely on your use case. If you need to know, use, or keep track of the position of an element within a sequence (e.g. *"Find the x-th element of the string"*), then using indices is the way to go.

Finally, the `while`-loop version will look as follows:

```python
first_string = input("Enter a string: ")
second_string = input("Enter a second string: ")

second_string_length = len(second_string)
index = 0

while index < second_string_length:
    letter = second_string[index]
    if letter in first_string:
        print(letter)

    index += 1
```

Check out the code here **membership.py** with the configuration `Membership`.

Part 2: *The* `str` *Object*

We've been spending a fair amount of time using strings now, so we should probably dive in a little deeper into their behavior.

Objects in general (not just `str` objects) have a set of values and operations associated to them. In technical jargon, we would say that objects have **reference attributes bound to them**. The values associated with objects are called **attributes** (similar to variables), and the operations associated with objects are called **methods** (similar to functions):

> > **Attribute**: A *variable* associated to an object of a certain type/class.
>
> > **Method**: A *function* associated to an object of a certain type/class.

Both of these terms will make more sense when we get to object-oriented programming, but just learn to recognise this behavior when I say things like "this is a *method* of a `str` object", for example.

Let's take a look at some examples:

```
>>> string = "the tale of the heike"
>>> string.__doc__
"str(object='') -> str\nstr(bytes_or_buffer[, encoding[, errors]]) -> str\n\nCrea
>>> string.__class__
<class 'str'>
```

In this case, `__doc__` is an **attribute** associated to the string object `string`, which prints out a bunch of technical information that you don't need to worry about. `__class__` is another such attribute, which tells us the name of the class of the current object (in this case, it's a `str` object).

Let's take a look now at a couple of **methods**:

```
>>> string = "the tale of the heike"
>>> string.capitalize()
'The tale of the heike'
>>> string.upper()
'THE TALE OF THE HEIKE'
```

In this case, `capitalize()` is ***returns*** a string with its first character capitalized (if the first character is not a lower-case letter, it will simply return the same string). By the same token, `upper()` ***returns*** a string with the original string's letters capitalized.

This is important. **Strings are immutable**; none of their methods that appear to be mutating them in any way, but rather creating an entirely new string object based on the contents of the original:

```
>>> example = "Mineral water"
>>> example.upper()
'MINERAL WATER'
>>> example
'Mineral water'
>>> example_upper = example.upper()
>>> example_upper
'MINERAL WATER'
```

The value of example didn't change when its `upper()` method was invoked.

Here are some examples of methods from the `str` class that we'll be making heavy use of in this class:

Method	Example	Description
`find()`	`"abc".find("a")` returns `0` `"abc".find("z")` returns `-1`	Returns the index of **first** occurrence of the argument in a string object.
`format()`	`"Hi, my name is {}, and I love {}.".format("Sebastian", "Zelda")` returns `"Hi, my name is Sebastian, and I love Zelda."`	Formats string into nicer output.
`isalnum()`	`"1978".isalpha()` returns `False` `"The1975".isalpha()` returns `True`	Returns `True` if all the characters in a `str` object are alphanumeric.
`isalpha()`	`"1978".isalpha()` returns `False` `"TheNineteenSeventyFive".isalpha()` returns `True`	Returns `True` if all the characters in a `str` object are alphabetic.

Method	Example	Description
`isdigit()`	`"1978".isdigit()` returns `False` `"The1975".isalpha()` returns `False`	Returns `True` if all the characters in a `str` object are numeric.
`islower()`	`"The The".islower()` returns `False`	Returns `True` if all the characters in a `str` object are lowercase.
`isupper()`	`"NYU".isupper()` returns `True`	Returns `True` if all the characters in a `str` object are uppercase.
`lower()`	`"Liz and The Blue Bird.".lower()` returns `"liz and the blue bird."`	Returns a `str` object with all uppercase characters from the original string lower-cased.
`upper()`	`"Liz and The Blue Bird.".upper()` returns `"LIZ AND THE BLUE BIRD."`	Returns a `str` object with all lowercase characters from the original string upper-cased.

Figure 1*: Some useful* `str` *methods in this class (****Full list****).*

There are obviously...a lot of them. But you don't have to memorize all of them. They're all pretty intuitive and self- explanatory, so it's actually really easy to remember them.

NOTE: You may ***not*** yet use `split()` or any other method that returns a `str` object into a `list` object or any other object that we haven't seen yet. This will be penalized in homework, so please always ask if you are unsure whether you are allowed to use something or not.

Part 3: *String Operations*

Concatenation

Okay, so if you can't modify strings, but your program requires you to put a bunch of strings together, what do we do?

Python makes this extremely easy for us, actually. We can quite literally add two strings together by using the `+` operator. This process is called **string concatenation**:

```
first_name = "Camille"
last_name = "Pissarro"
full_name = first_name + " " + last_name

print(full_name)
```

Output:

```
Camille Pissarro
```

Nothing we haven't seen before. We just have a name for it now. Again, remember neither `first_name` nor `last_name` are being modified in any way here. All Python is doing here is extracting the contents of `first_name` and `last_name` and creating a completely new string out of them, `full_name`.

What about this example?

```
first_name = "Camille"
last_name = "Pissarro"
full_name = first_name + " " + last_name
age = 73
full_info = full_name + ", " + age

print(full_info)
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 5, in <module>
TypeError: can only concatenate str (not "int") to str
```

Ah-hah. We've seen this error before. You cannot concatenate a `str` and an `int`. So if you want to create a new string from these components, you have to **explicitly convert all non-`str` components into `str` objects**:

```
first_name = "Camille"
last_name = "Pissarro"
full_name = first_name + " " + last_name
```

```
age = 73
full_info = full_name + ", " + str(age)

print(full_info)
```

Output

```
Camille Pissarro, 73
```

We didn't have to do this when we were composing `print()` statements because `print()` takes care fo the type conversions for us. But more often than not, we will not be able to rely on `print()` so it's important to know how to do it yourself.

Slicing

The second important operation that we can do with strings is *slicing* them—that is, extracting a specified subsection.

We, of course, already learned how to index a string, which is technically a form of slicing:

```
>>> "Ahmad Jamal"[4]
'd'
```

But, just like with the `range()` function, we have the power to define our starting, ending, and stepping value using the following syntax:

```
>>> "Ahmad Jamal"[2:7]
'mad J'
```

```
>>> jazz_musician = "Ahmad Jamal"
>>> jazz_musician[2:len(jazz_musician):2]
'mdJml'
```