Python programming for beginners

Stefan Zhauryd
Instructor

Module 6

Strings, String Methods, Exceptions

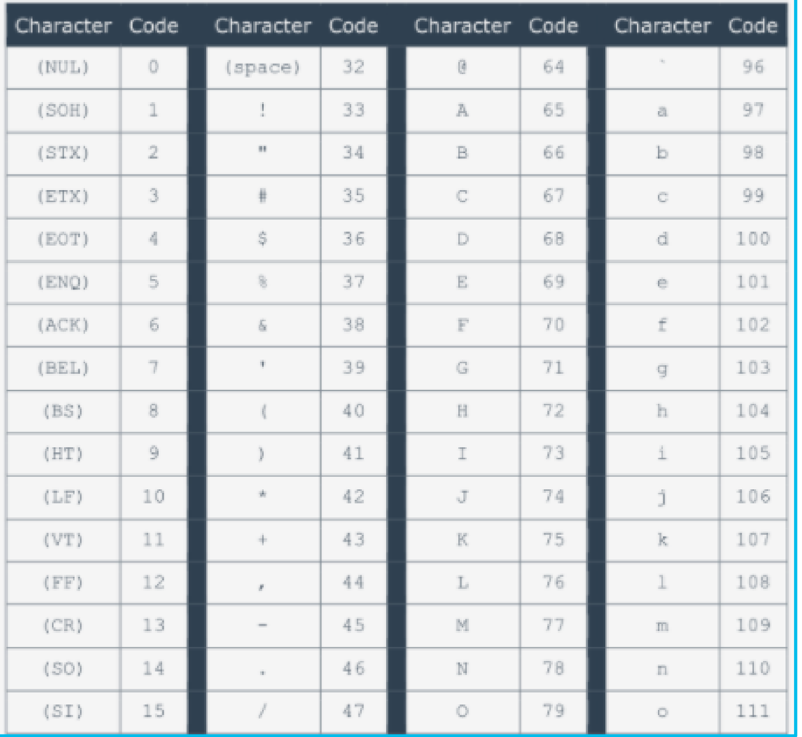
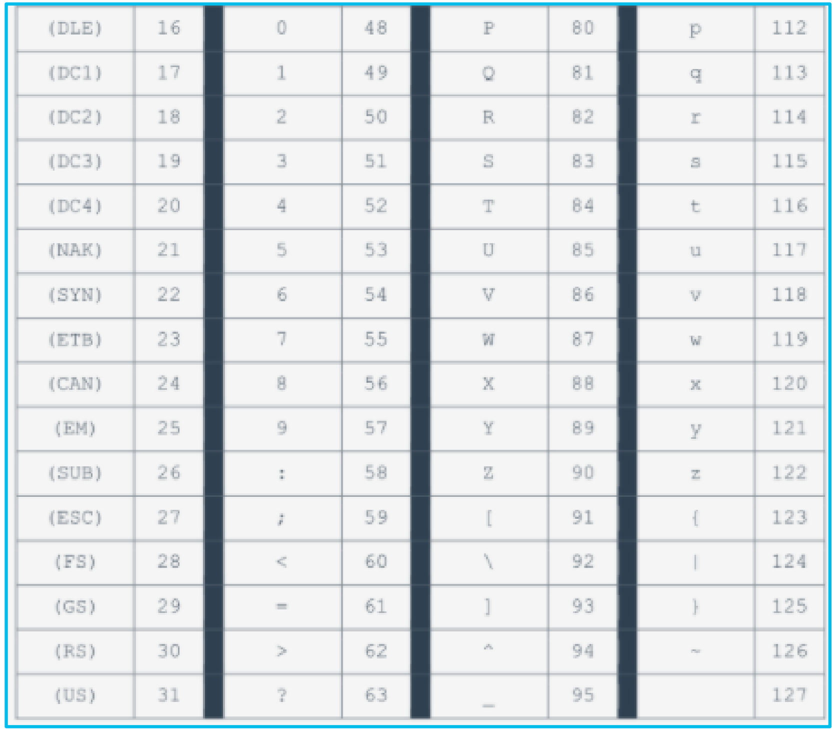
In this module, you will learn about:

- Characters, strings and coding standards;
- Strings vs. lists – similarities and differences;
- Lists methods;
- String methods;
- Python's way of handling runtime errors;
- Controlling the flow of errors using try and except;
- Hierarchy of exceptions.

How computers understand single characters

Computers store characters as numbers. Every character used by a computer corresponds to a unique number, and vice versa. This assignment must include more characters than you might expect. Many of them are invisible to humans, but essential to computers.

Some of these characters are called whitespaces, while others are named control characters, because their purpose is to control input/output devices.

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	53	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119
(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

ASCII

(short for **American Standard Code for Information Interchange**)

The code provides space for 256 different characters, but we are interested only in the first 128. If you want to see how the code is constructed, look at the table below. Click the table to enlarge it. Look at it carefully - there are some interesting facts. Look at the code of the most common character - the space. This is 32.

https://ru.wikipedia.org/wiki/ASCII

I18N

I18N

INTERNATIONALIZATION

A classic form of ASCII code uses eight bits for each sign. Eight bits mean 256 different characters. The first 128 are used for the standard Latin alphabet (both upper-case and lower-case characters). Is it possible to push all the other national characters used around the world into the remaining 128 locations?

Code points and code pages

A code point is a number which makes a character. For example, **32** is a code point which makes a **space** in **ASCII** encoding. We can say that standard **ASCII** code consists of **128** code points.

As standard **ASCII** occupies **128** out of **256** possible code points, you can only make use of the remaining **128**.

It's not enough for all possible languages, but it may be sufficient for one language, or for a small group of similar languages.

Can you set the higher half of the code points differently for different languages?

Code points and **code pages**

A code page is a standard for using the upper **128** code points to store specific national characters. For example, there are different code pages for Western Europe and Eastern Europe, Cyrillic and Greek alphabets, Arabic and Hebrew languages, and so on.

This means that the one and same code point can make different characters when used in different code pages.

For example, the code point **200** makes **Č** (a letter used by some **Slavic languages**) when utilized by the **ISO/IEC 8859-2** code page, and makes **Ш** (a **Cyrillic letter**) when used by the **ISO/IEC 8859-5** code page.

Unicode

https://www.ibm.com/docs/en/i/7.3?topic=applications-working-unicode
https://habr.com/ru/company/vk/blog/547084/

Unicode assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points. The first **128 Unicode code points** are identical to **ASCII**, and the first **256 Unicode code points** are identical to the **ISO/IEC 8859-1 code page** (a code page designed for western European languages).

UCS-4/UTF-32

UCS-4

32 bits to store each character

https://www.oxfordreference.com/view/10.1093/oi/authority.20110803110450734

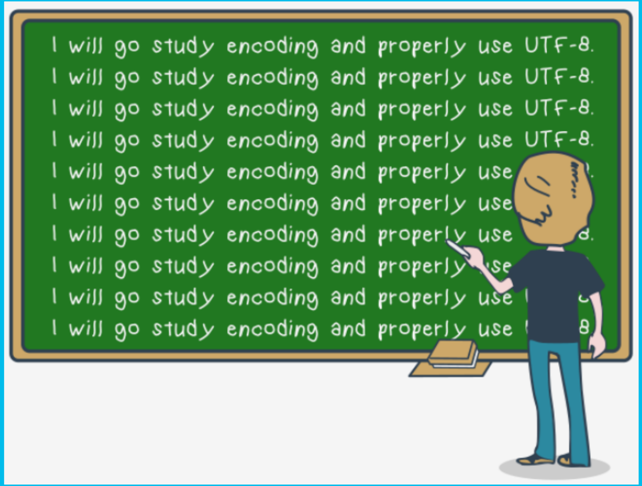
The Unicode standard says nothing about how to code and store the characters in the memory and files. It only names all available characters and assigns them to planes (a group of characters of similar origin, application, or nature).

There is more than one standard describing the techniques used to implement Unicode in actual computers and computer storage systems. The most general of them is **UCS-4**.

The name comes from **Universal Character Set**.

UCS-4 uses **32 bits** (four bytes) to store each character, and the code is just the Unicode code points' unique number. A file containing **UCS-4** encoded text may start with a **BOM (byte order mark)**, an unprintable combination of bits announcing the nature of the file's contents. Some utilities may require it.

UTF-8


I will go study encoding and properly use UTF-8.
I will go study encoding and properly use UTF-8.
I will go study encoding and properly use UTF-8.
I will go study encoding and properly use UTF-8.

One of the most commonly used is **UTF-8**.

The name is derived from **Unicode Transformation Format**.

The concept is very smart. **UTF-8** uses as many bits for each of the code points as it really needs to represent them.

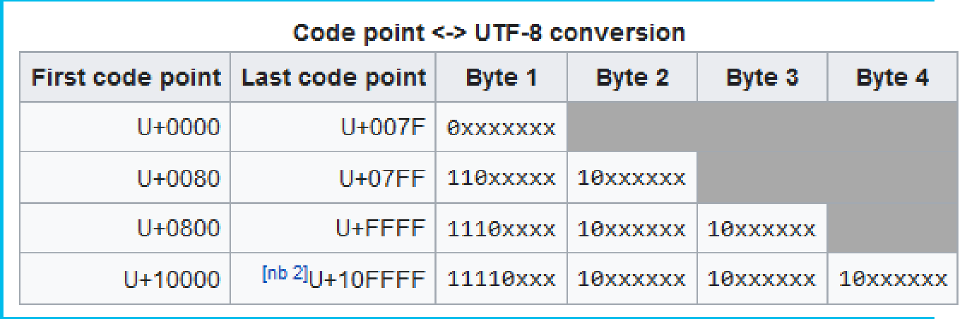
https://unicode.org/faq/utf_bom.html

https://www.ibm.com/docs/en/i/7.3?topic=unicode-utf-8

https://ru.wikipedia.org/wiki/UTF-8

https://en.wikipedia.org/wiki/UTF-8

UTF-8



Code point <-> UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[nb 2]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

For example:

- **all Latin characters** (and all standard ASCII characters) occupy **eight bits (8 bits)**;
- **non-Latin** characters occupy **16 bits**;
- **CJK** (China-Japan-Korea) ideographs occupy **24 bits**.

Due to features of the method used by **UTF-8** to store the code points, there is no need to use the **BOM**, but some of the tools look for it when reading the file, and many editors set it up during the save.

Python 3 fully supports Unicode and UTF-8:

- **you can use Unicode/UTF-8 encoded characters** to name variables and other entities;
- **you can use them during all input and output.**

This means that **Python3 is completely I18Ned**.

Key takeaways

1. Computers **store characters as numbers**. There is more than one possible way of encoding characters, but only some of them gained worldwide popularity and are commonly used in IT: these are

- **ASCII** (**used mainly to encode the Latin alphabet and some of its derivates**) and
- **UNICODE** (**able to encode virtually all alphabets being used by humans**).

2. **A number corresponding to a particular character is called a codepoint**.

3. **UNICODE** uses different ways of encoding when it comes to storing the characters using files or computer memory: **two of them are UCS-4(UTF-32)** and **UTF-8** (the latter is the most common as it wastes less memory space).

Exercises

- What is BOM?
- What is the I18N?
- Is Python 3 I18Ned?

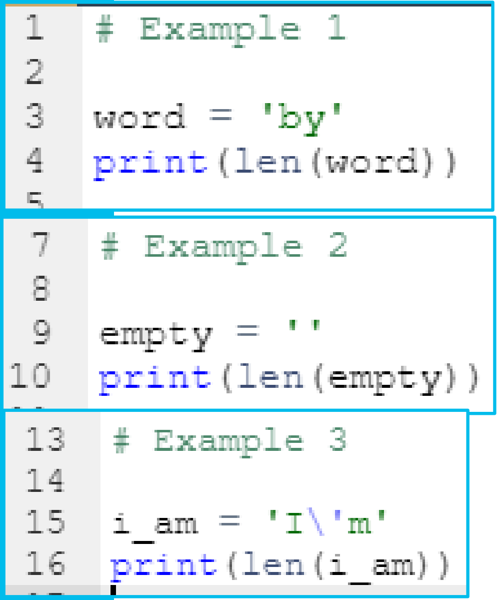
- What is the ASCII?
- What is the Unicode?
- What is the UCS-4?

- What is the UTF-8?

Strings - a brief review

First of all, Python's strings (or simply strings, as we're not going to discuss any other language's strings) are **immutable sequences**.

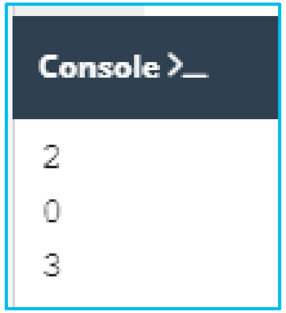
Let's analyze the code in the editor to understand what we're talking about:



```python
# Example 1

word = 'by'
print(len(word))
```

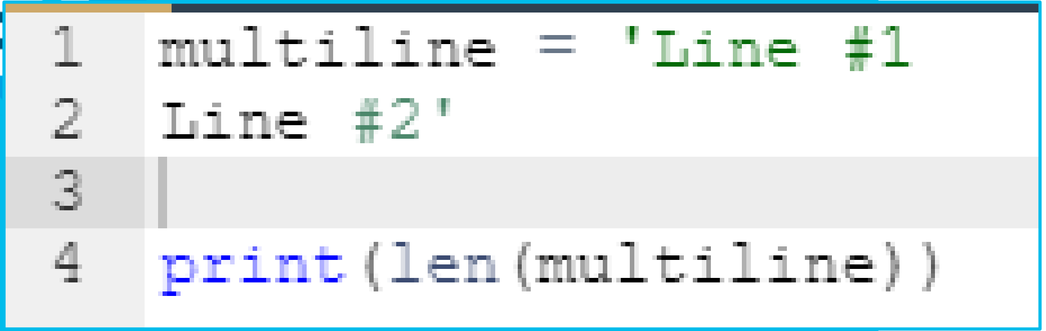
```python
# Example 2

empty = ''
print(len(empty))
```

```python
# Example 3

i_am = 'I\'m'
print(len(i_am))
```

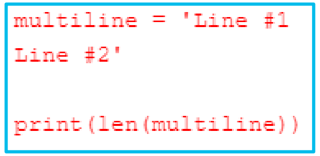


Console >_

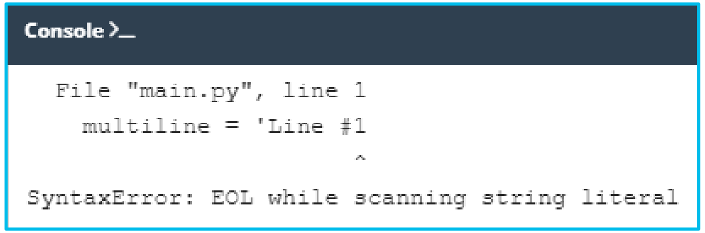
```
2
0
3
```



```
multiline = 'Line #1
Line #2'

print(len(multiline))
```

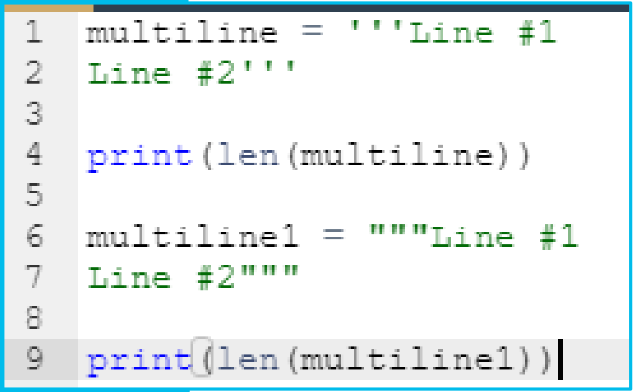


```
multiline = 'Line #1
Line #2'

print(len(multiline))
```

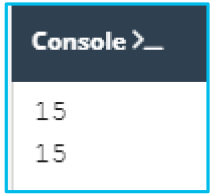


```
  File "main.py", line 1
    multiline = 'Line #1
                        ^
SyntaxError: EOL while scanning string literal
```

Multiline strings



```
multiline = '''Line #1
Line #2'''

print(len(multiline))

multiline1 = """Line #1
Line #2"""

print(len(multiline1))
```



```
15
15
```

Operations on strings

In general, **strings can be**:

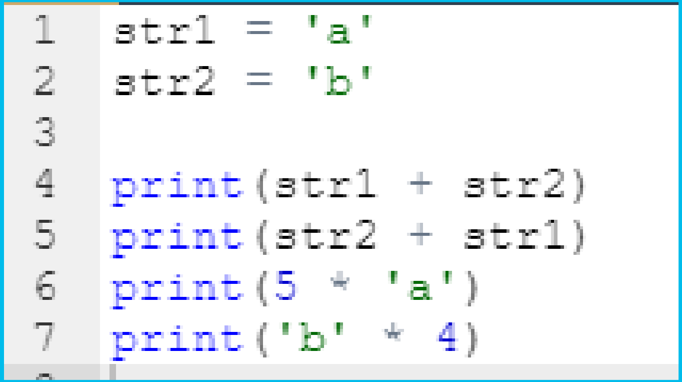
- **concatenated (joined)**
- **replicated.**

The first operation is performed by the **+ operator** (**note: it's not an addition**) while the second by the *** operator** (**note again: it's not a multiplication**).

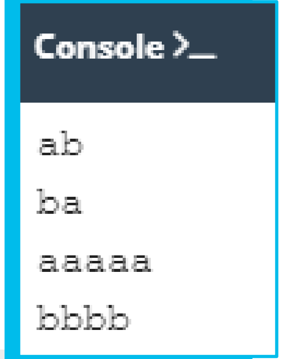
*The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called **overloading** (as such an operator is overloaded with different duties).*

Note: shortcut variants of the above operators are also applicable for strings (+= and *=).

Operations on strings



```
str1 = 'a'
str2 = 'b'

print(str1 + str2)
print(str2 + str1)
print(5 * 'a')
print('b' * 4)
```



```
ab
ba
aaaaa
bbbb
```

Analyze the example:

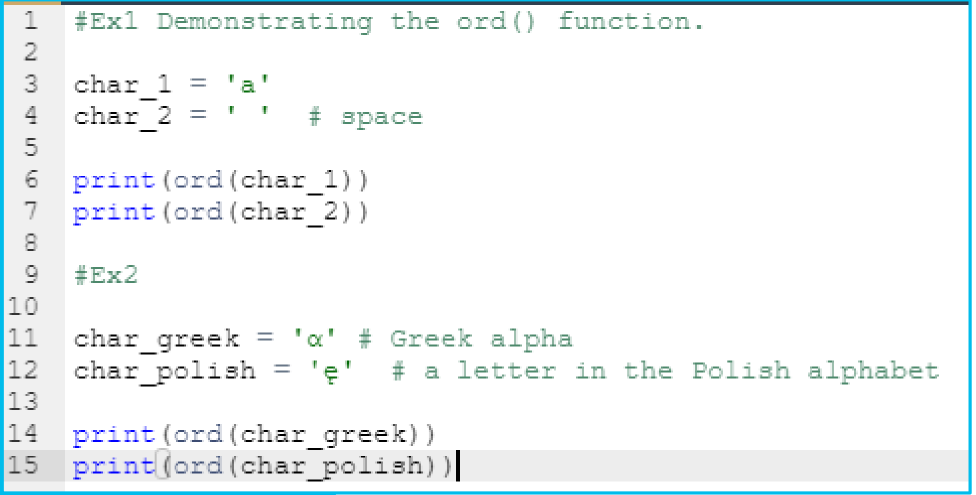
- The + operator used against two or more strings produces a new string containing all the characters from its arguments (note: the order matters - this overloaded +, in contrast to its numerical version, is not commutative)
- the * operator needs a string and a number as arguments; in this case, the order doesn't matter - you can put the number before the string, or vice versa, the result will be the same - a new string created by the nth replication of the argument's string.

The snippet produces the following output:

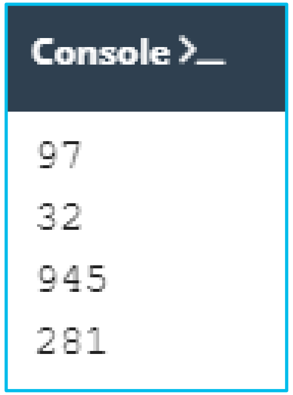
Operations on strings: **ord()**

If you want to know a specific character's **ASCII/UNICODE** code point value, you can use a function named **ord()** (as in ordinal).

The function needs a **one-character string** as **its argument** - breaching this requirement causes a **TypeError exception**, and **returns a number** representing the argument's code point.

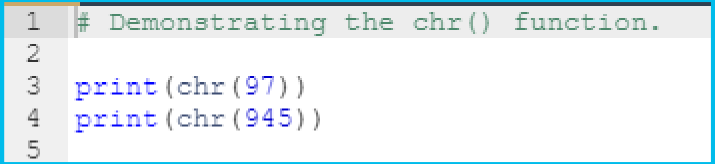


```
#Ex1 Demonstrating the ord() function.

char_1 = 'a'
char_2 = ' '  # space

print(ord(char_1))
print(ord(char_2))

#Ex2

char_greek = 'α' # Greek alpha
char_polish = 'ę'  # a letter in the Polish alphabet

print(ord(char_greek))
print(ord(char_polish))
```

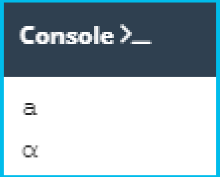


Console >_

```
97
32
945
281
```



```python
# Demonstrating the chr() function.

print(chr(97))
print(chr(945))

```



```
Console >_
a
α
```

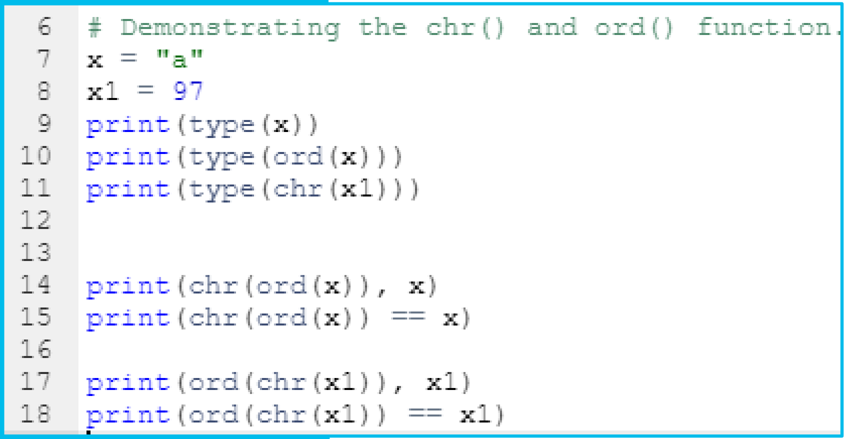
Operations on strings: **chr()**

If you know the code point (number) and want to get the corresponding character, you can use a function named chr().

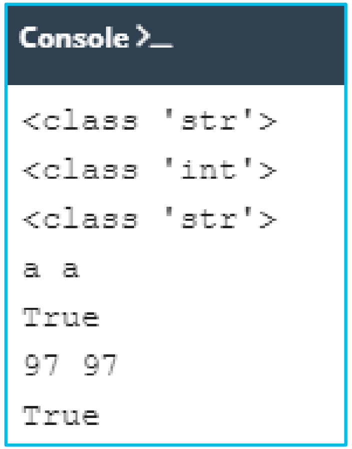
The function **takes a code point** and **returns its character**.

Invoking it with an invalid argument (e.g., a negative or invalid code point) causes ValueError or TypeError exceptions.

```
for i in range(1000):
        print(chr(i))
```

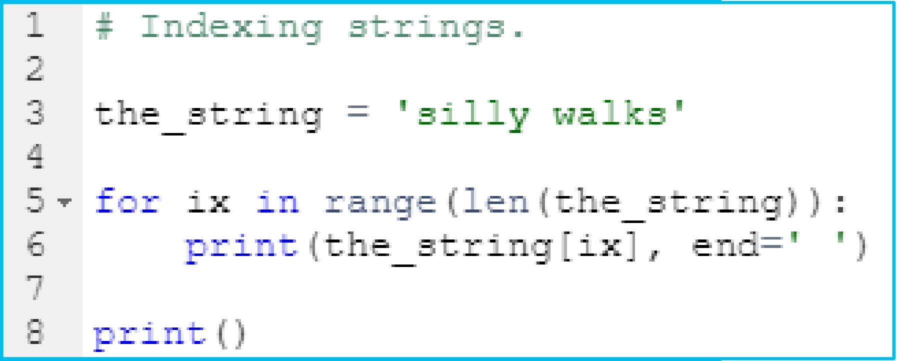


```python
# Demonstrating the chr() and ord() function.
x = "a"
x1 = 97
print(type(x))
print(type(ord(x)))
print(type(chr(x1)))


print(chr(ord(x)), x)
print(chr(ord(x)) == x)

print(ord(chr(x1)), x1)
print(ord(chr(x1)) == x1)
```

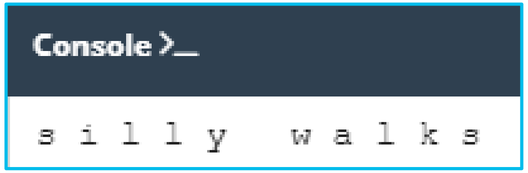


```
Console >_
<class 'str'>
<class 'int'>
<class 'str'>
a a
True
97 97
True
```

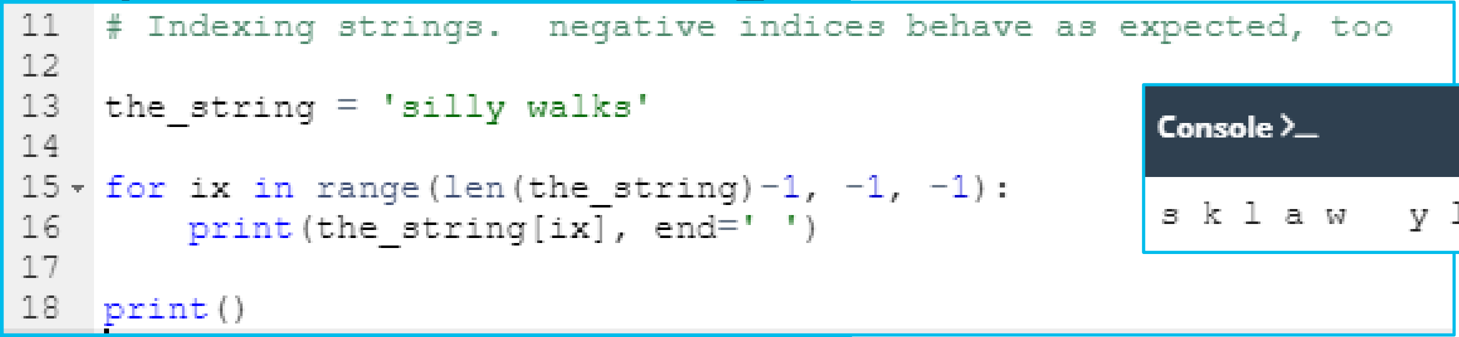
Strings as sequences: indexing



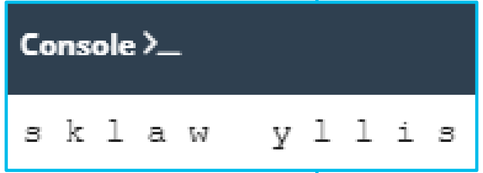
```python
# Indexing strings.

the_string = 'silly walks'

for ix in range(len(the_string)):
    print(the_string[ix], end=' ')

print()
```



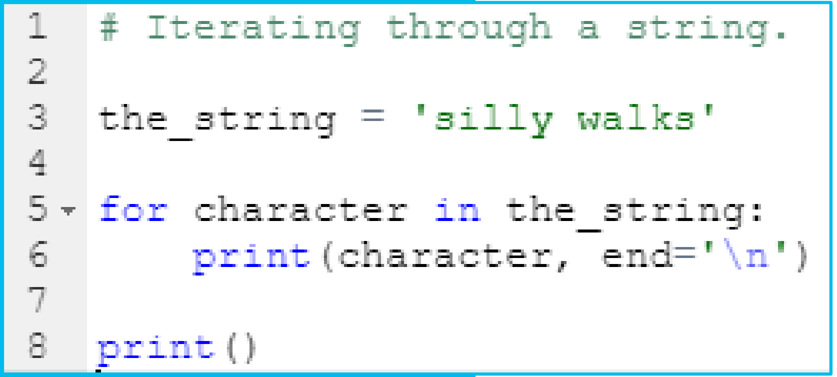
```
Console >_
s i l l y   w a l k s
```



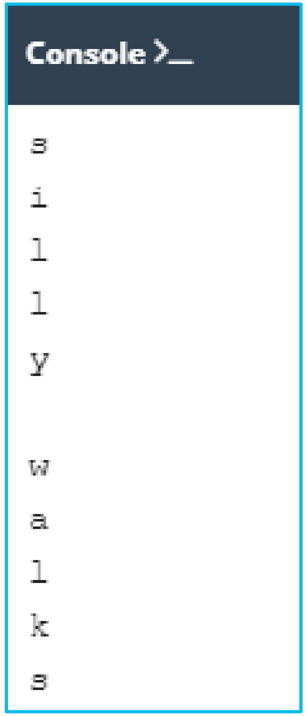
```python
# Indexing strings.  negative indices behave as expected, too

the_string = 'silly walks'

for ix in range(len(the_string)-1, -1, -1):
    print(the_string[ix], end=' ')

print()
```



```
Console >_
s k l a w   y l l i s
```



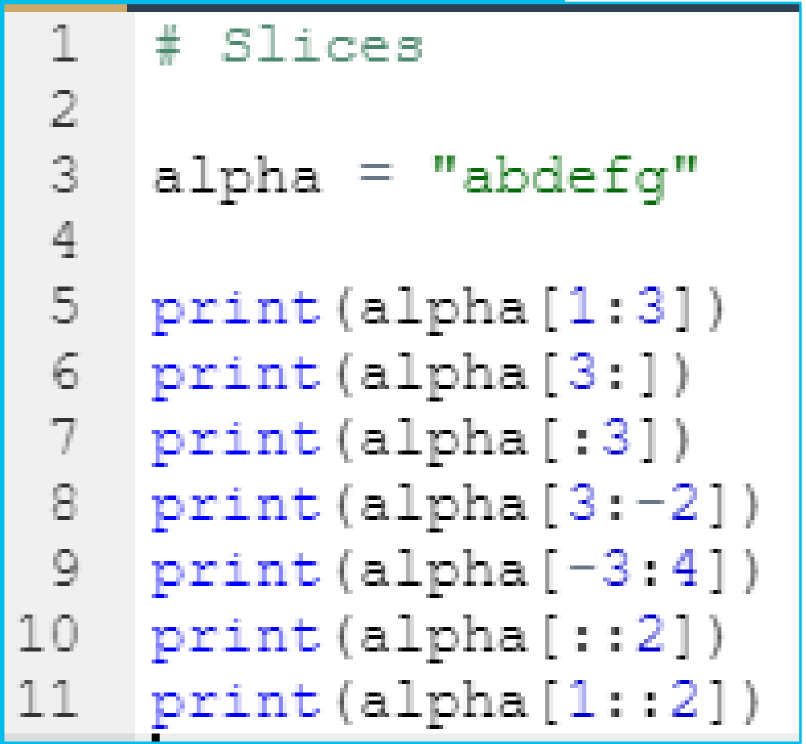
```
# Iterating through a string.

the_string = 'silly walks'

for character in the_string:
    print(character, end='\n')

print()
```



```
s
i
l
l
y

w
a
l
k
s
```

Strings as sequences: iterating

Slices



```
1  # Slices
2
3  alpha = "abdefg"
4
5  print(alpha[1:3])
6  print(alpha[3:])
7  print(alpha[:3])
8  print(alpha[3:-2])
9  print(alpha[-3:4])
10 print(alpha[::2])
11 print(alpha[1::2])
```

S = a[:2] + ‘lol’ + a[3:]

‘ab’ + ‘lol’ + ‘efg’

8 - **e**

9 - **e**

10 - **adf**

11 - **beg**

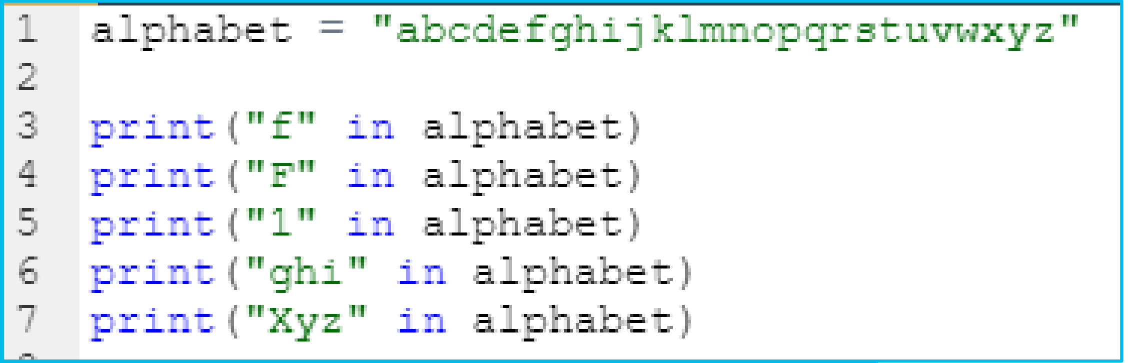
The **in** and **not in** operators

The in operator

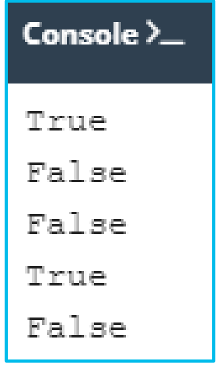
The in operator shouldn't surprise you when applied to strings - it simply checks if its left argument (a string) can be found anywhere within the right argument (another string).

The result of the check is simply True or False.

Look at the example program below. This is how the in operator works:



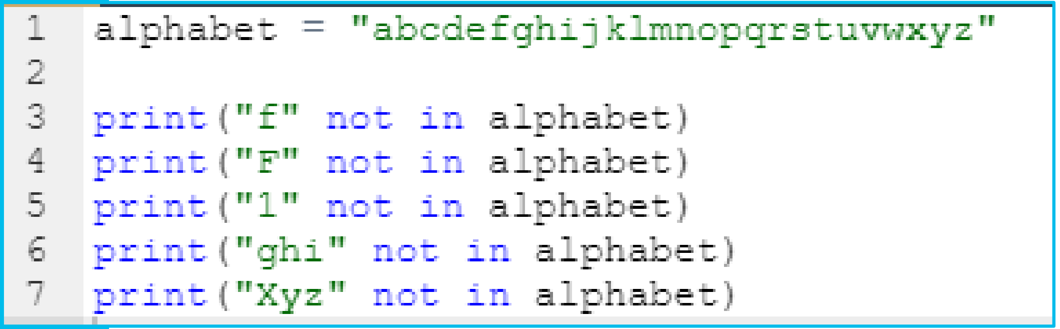
```python
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)
print("F" in alphabet)
print("1" in alphabet)
print("ghi" in alphabet)
print("Xyz" in alphabet)
```



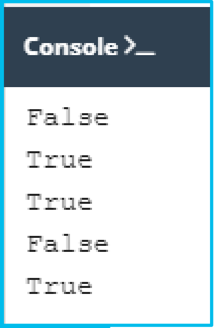
```
True
False
False
True
False
```

The **in** and **not in** operators

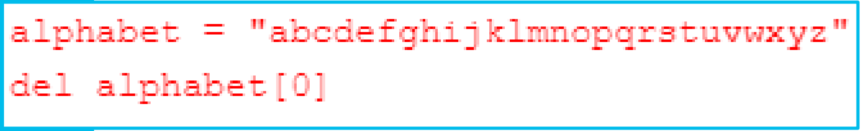
The not in operator



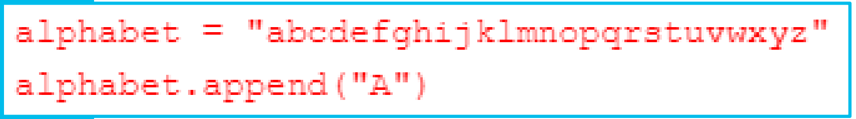
```python
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" not in alphabet)
print("F" not in alphabet)
print("1" not in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```



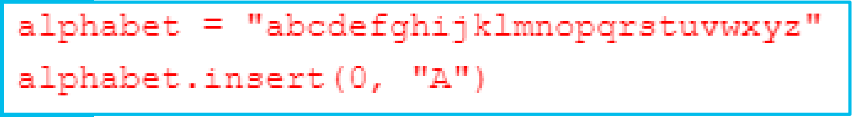
```
False
True
True
False
True
```



```python
alphabet = "abcdefghijklmnopqrstuvwxyz"
del alphabet[0]
```

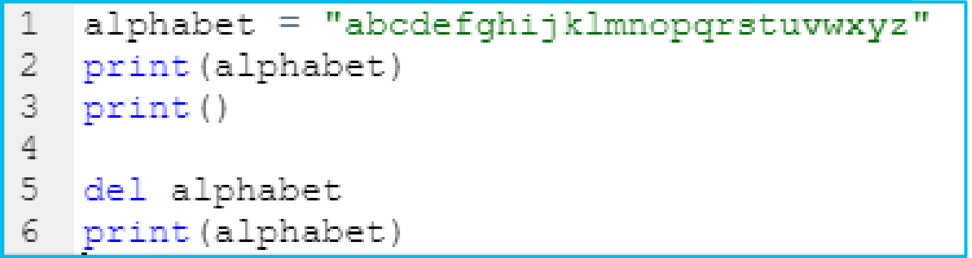


```python
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.append("A")
```

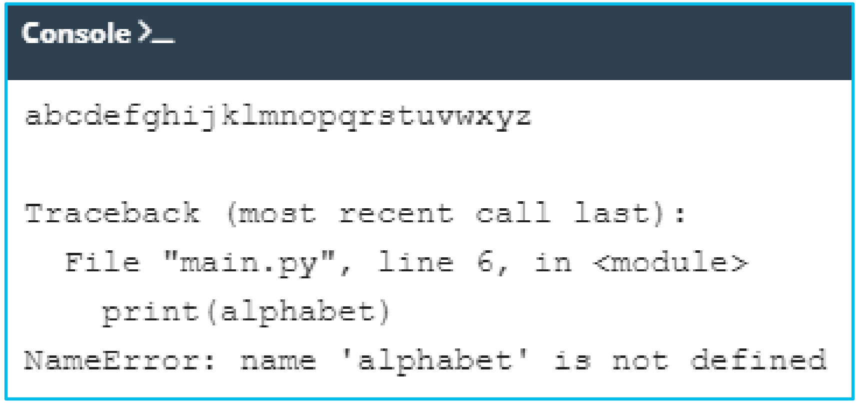


```python
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.insert(0, "A")
```

Python strings are immutable



```python
alphabet = "abcdefghijklmnopqrstuvwxyz"
print(alphabet)
print()

del alphabet
print(alphabet)
```

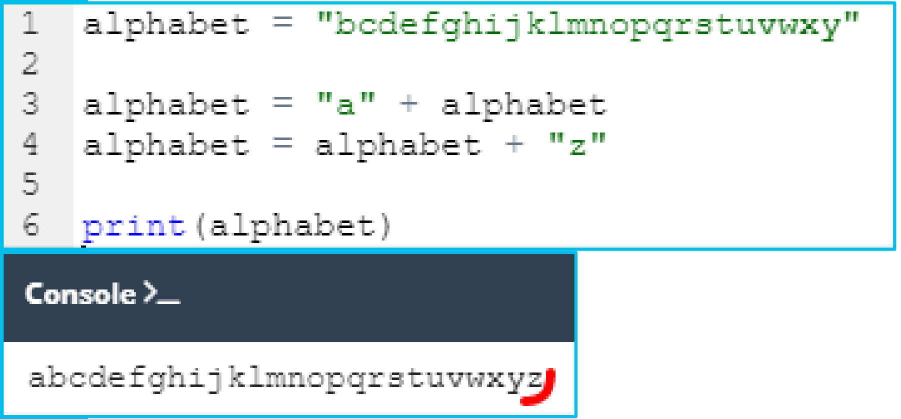


Console >_

```
abcdefghijklmnopqrstuvwxyz


Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(alphabet)
NameError: name 'alphabet' is not defined
```

Operations on strings: continued

This form of code is fully acceptable, will work without bending Python's rules, and will bring the full Latin alphabet to your screen:

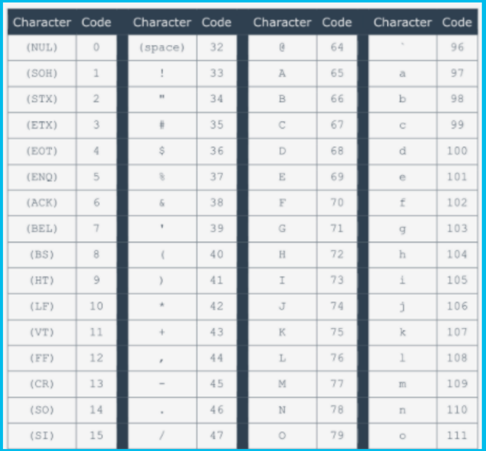


```python
alphabet = "bcdefghijklmnopqrstuvwxy"

alphabet = "a" + alphabet
alphabet = alphabet + "z"

print(alphabet)
```

Console >_

```
abcdefghijklmnopqrstuvwxyz
```

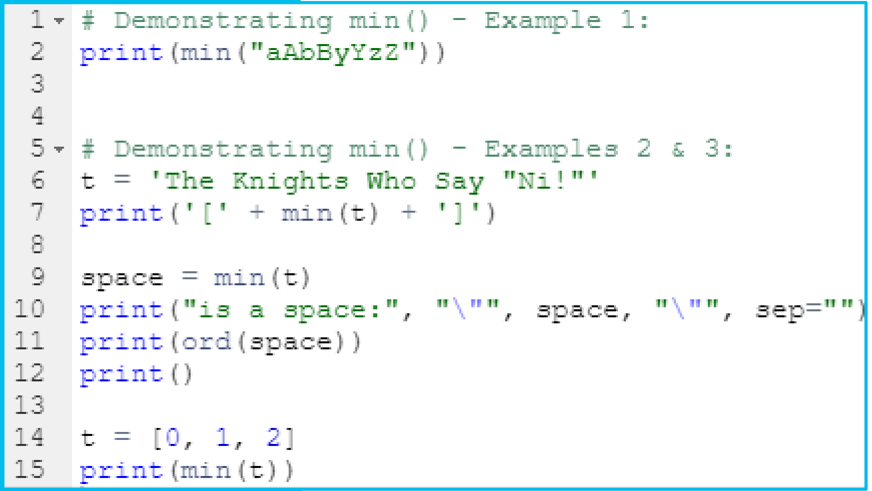
You may want to ask if creating a new copy of a string each time you modify its contents worsens the effectiveness of the code.



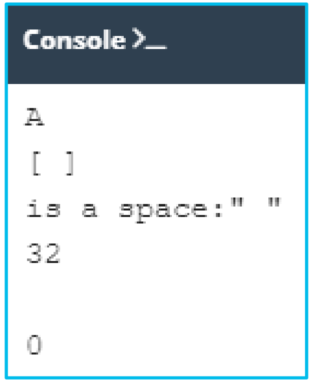
Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111

Operations on strings: **min()**

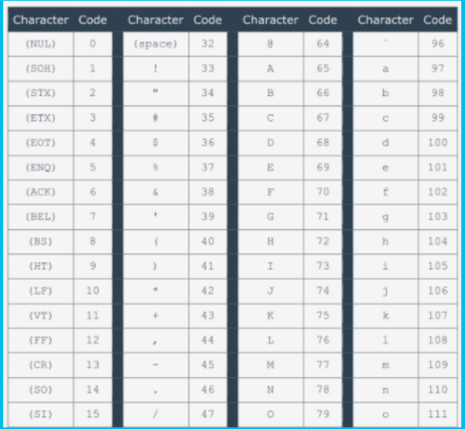
The function finds the **minimum** element of the sequence passed as an argument. There is one condition - the sequence (string, list, it doesn't matter) cannot be empty, or else you'll get a **ValueError** exception.



```python
# Demonstrating min() - Example 1:
print(min("aAbByYzZ"))


# Demonstrating min() - Examples 2 & 3:
t = 'The Knights Who Say "Ni!"'
print('[' + min(t) + ']')

space = min(t)
print("is a space:", "\"", space, "\"", sep=""
print(ord(space))
print()

t = [0, 1, 2]
print(min(t))
```



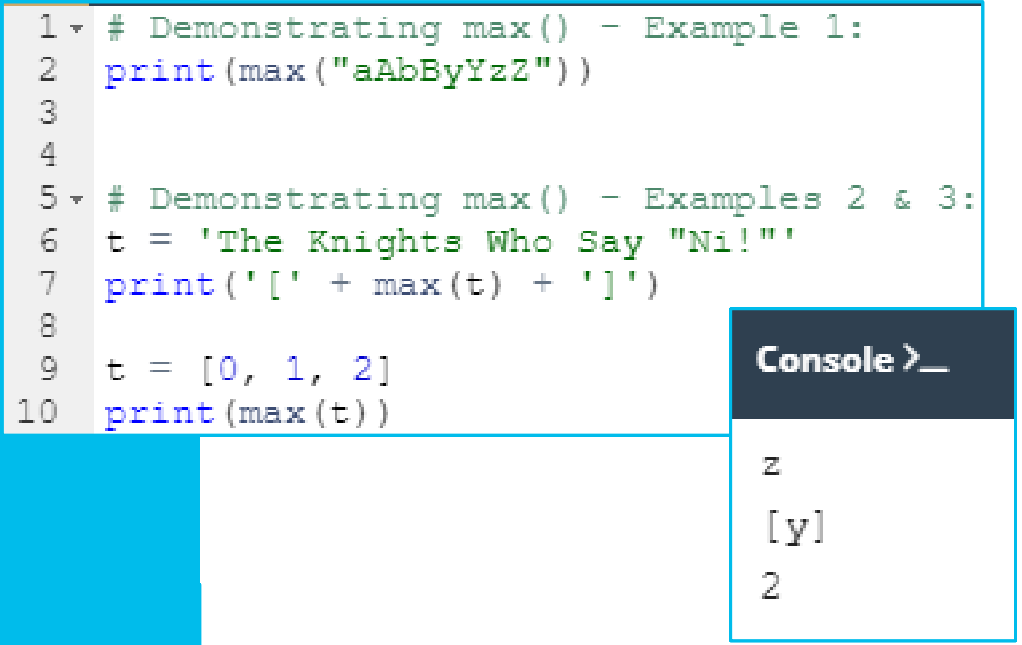
```
A
[ ]
is a space:" "
32

0
```



Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111

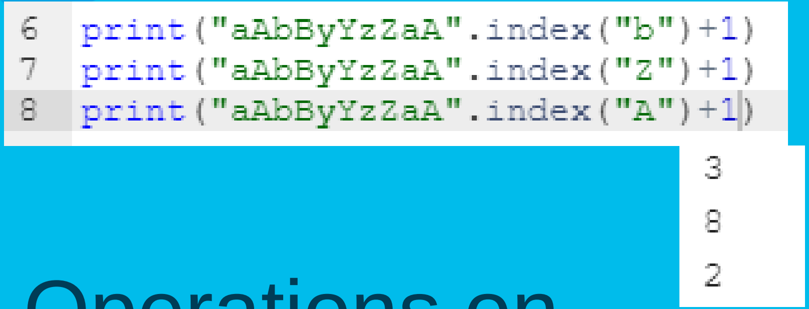
Operations on strings: **max()**

Similarly, a function named max() finds the **maximum** element of the sequence.



```python
# Demonstrating max() - Example 1:
print(max("aAbByYzZ"))


# Demonstrating max() - Examples 2 & 3:
t = 'The Knights Who Say "Ni!"'
print('[' + max(t) + ']')

t = [0, 1, 2]
print(max(t))
```

```
z
[y]
2
```



```
print("aAbByYzZaA".index("b")+1)
print("aAbByYzZaA".index("Z")+1)
print("aAbByYzZaA".index("A")+1)
```

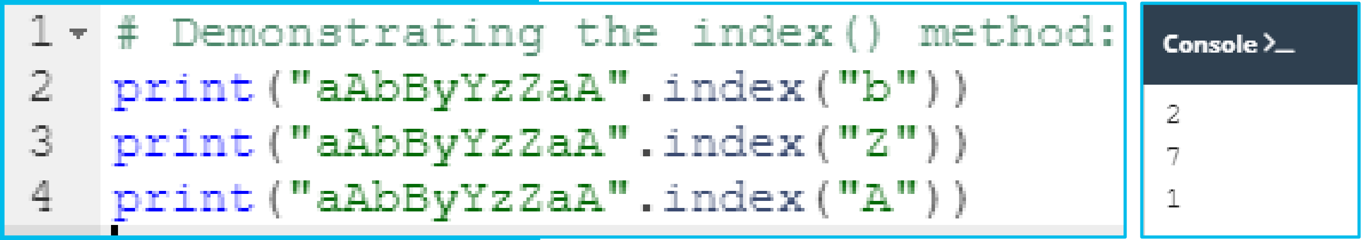
```
3
8
2
```

Operations on strings: the **index()** method

The index() method (it's a method, not a function) **searches the sequence from the beginning**, in order to find the first element of the value specified in its argument.

Note: the element searched for must occur in the sequence - its absence will cause a **ValueError** exception.

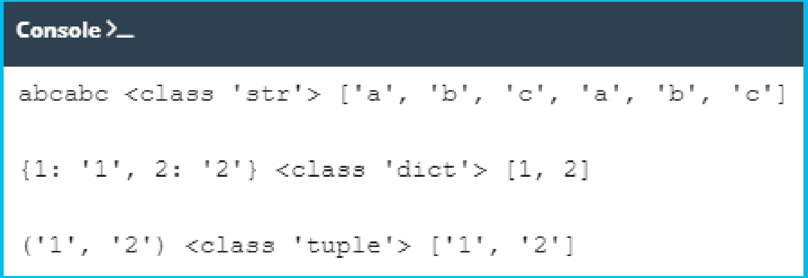
The method r**eturns the index of the first occurrence of the argument** (which means that the lowest possible result is 0, while the highest is the length of argument decremented by 1).



```
# Demonstrating the index() method:
print("aAbByYzZaA".index("b"))
print("aAbByYzZaA".index("Z"))
print("aAbByYzZaA".index("A"))
```

```
2
7
1
```

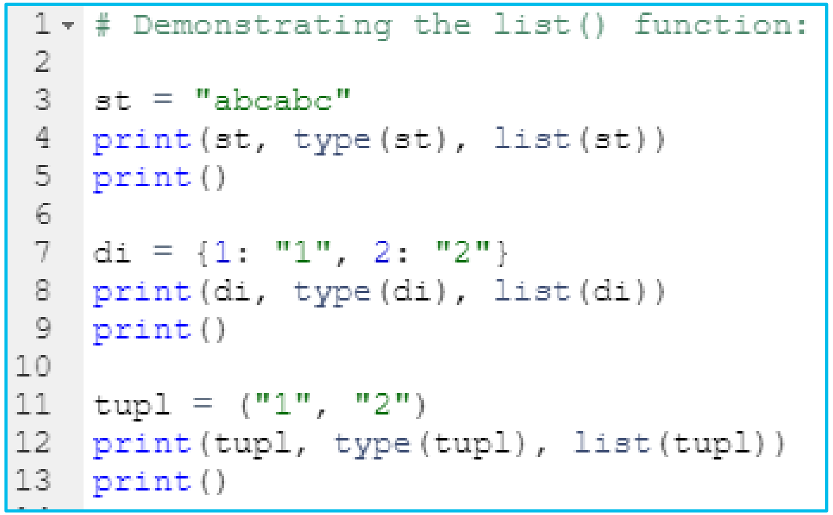
Operations on strings: the **list()** function



```
Console >_

abcabc <class 'str'> ['a', 'b', 'c', 'a', 'b', 'c']

{1: '1', 2: '2'} <class 'dict'> [1, 2]

('1', '2') <class 'tuple'> ['1', '2']
```

The list() function **takes its argument (a string)** and **creates a new list containing all the string's characters**, one per list element.

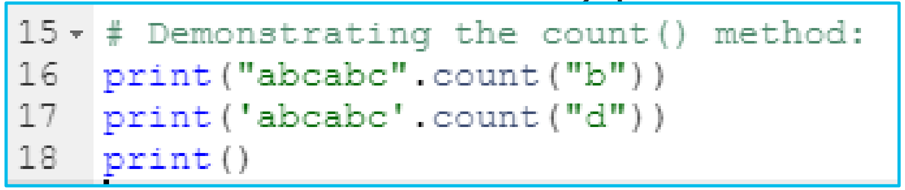
Note: it's not strictly a string function - **list() is able to create a new list from many other entities** (e.g., from **tuples** and **dictionaries**).



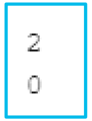
```python
# Demonstrating the list() function:

st = "abcabc"
print(st, type(st), list(st))
print()

di = {1: "1", 2: "2"}
print(di, type(di), list(di))
print()

tupl = ("1", "2")
print(tupl, type(tupl), list(tupl))
print()
```

Operations on strings: the **count()** method

The **count()** method **counts all occurrences of the element inside the sequence**. The absence of such elements doesn't cause any problems.



```python
# Demonstrating the count() method:
print("abcabc".count("b"))
print('abcabc'.count("d"))
print()
```



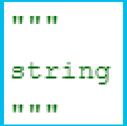
```
2
0
```

Python strings have a significant number of methods intended exclusively for processing characters. The complete list of is presented here: https://docs.python.org/3.4/library/stdtypes.html#string-methods

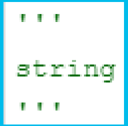
Key takeaways

1. Python strings are immutable sequences and can be indexed, sliced, and iterated like any other sequence, as well as being subject to the in and not in operators. There are two kinds of strings in Python:

- one-line strings, which cannot cross line boundaries – we denote them using either apostrophes ('string') or quotes ("string")
- multi-line strings, which occupy more than one line of source code, delimited by trigraphs:

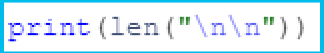


```
"""
string
"""
```



```
'''
string
'''
```

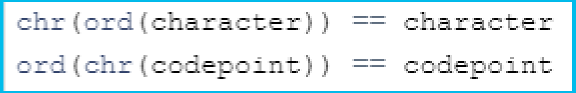
2. The length of a string is determined by the len() function. The escape character (\) is not counted. For example:



```
print(len("\n\n"))
```

3. Strings can be concatenated using the + operator, and replicated using the * operator.

4. The pair of functions chr() and ord() can be used to create a character using its codepoint, and to determine a codepoint corresponding to a character.



```
chr(ord(character)) == character
ord(chr(codepoint)) == codepoint
```

5. Some other functions that can be applied to strings are:
- list() – create a list consisting of all the string's characters;
- max() – finds the character with the maximal codepoint;
- min() – finds the character with the minimal codepoint.

6. The method named index() finds the index of a given substring inside the string.

Exercises

What is the length of the following string assuming there is no whitespaces between the quotes?

```
"""

"""
```

What is the expected output of the following code?

```
s = 'yesteryears'
the_list = list(s)
print(the_list[3:6])
```

What is the expected output of the following code?

```
for ch in "abc":
    print(chr(ord(ch) + 1), end='')
```

Answers

What is the length of the following string assuming there is no whitespaces between the quotes?

```
"""

"""
```

_1______________________________________

What is the expected output of the following code?

```
s = 'yesteryears'

the_list = list(s)

print(the_list[3:6])
```

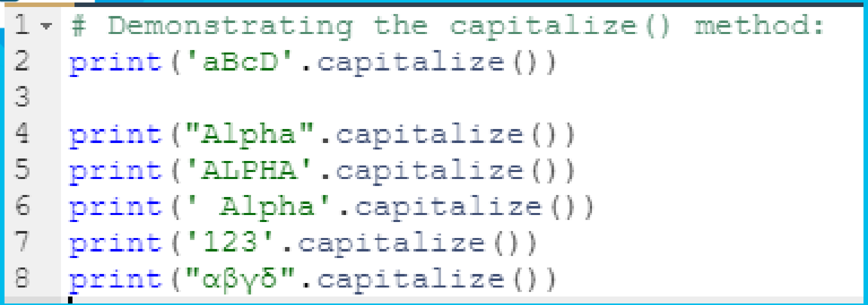
__['t', 'e', 'r']______________________________

What is the expected output of the following code?

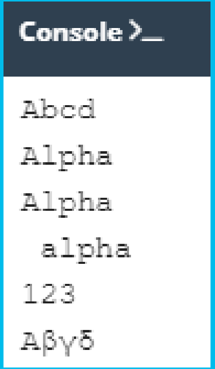
```
for ch in "abc":

    print(chr(ord(ch) + 1), end='')
```

_bcd____________________________________

The **capitalize()** method



```python
# Demonstrating the capitalize() method:
print('aBcD'.capitalize())

print("Alpha".capitalize())
print('ALPHA'.capitalize())
print(' Alpha'.capitalize())
print('123'.capitalize())
print("αβγδ".capitalize())
```



```
Console >_

Abcd
Alpha
Alpha
 alpha
123
Αβγδ
```

The **capitalize()** method does exactly what it says - **it creates a new string filled with characters taken from the source string**, but it tries to modify them in the following way:

- **if the first character inside the string is a letter** (note: the first character is an element with an index equal to 0, not just the first visible character), **it will be converted to upper-case**;
- **all remaining letters** from the string will be converted **to lower-case**.

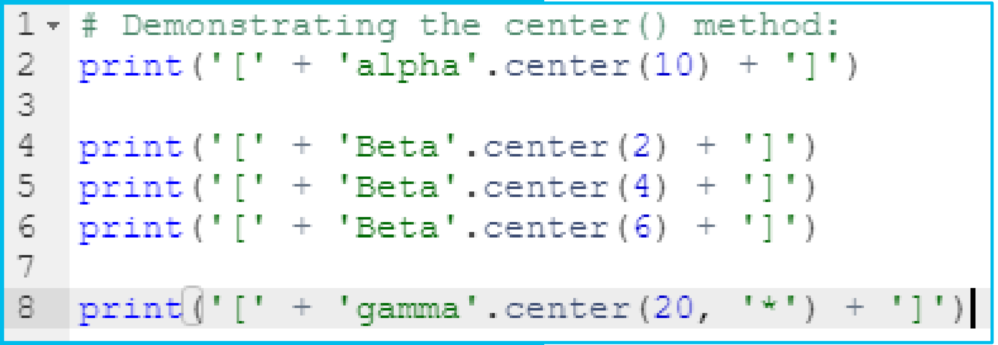
Note: methods don't have to be invoked from within variables only. **They can be invoked directly from within string literals**. We're going to use that convention regularly - it will simplify the examples, as the most important aspects will not disappear among unnecessary assignments.

The **center()** method

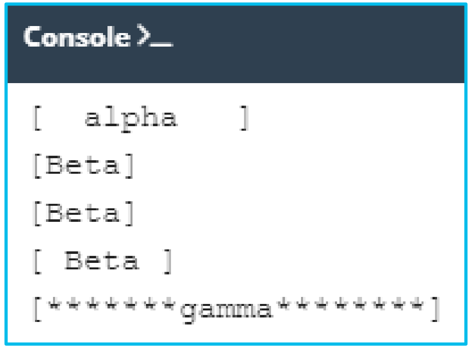
The **one-parameter variant** of the center() method **makes a copy of the original string**, trying to center it inside a field of a specified width.

The **centering** is actually done **by adding some spaces before and after** the string.

The **two-parameter variant** of center() makes **use of the character from the second argument**, **instead of a space**.



```python
# Demonstrating the center() method:
print('[' + 'alpha'.center(10) + ']')

print('[' + 'Beta'.center(2) + ']')
print('[' + 'Beta'.center(4) + ']')
print('[' + 'Beta'.center(6) + ']')

print('[' + 'gamma'.center(20, '*') + ']')
```



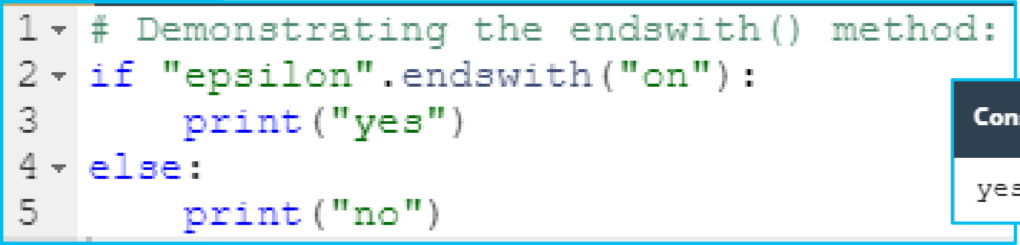
Console >_

```
[  alpha   ]
[Beta]
[Beta]
[ Beta ]
[*******gamma********]
```

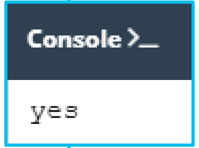
The **endswith()** method

The **endswith()** method **checks if the given string ends with the specified argument** and **returns True** or **False**, **depending on the check result**.

Note: the substring must adhere to the string's last character - it cannot just be located somewhere near the end of the string.

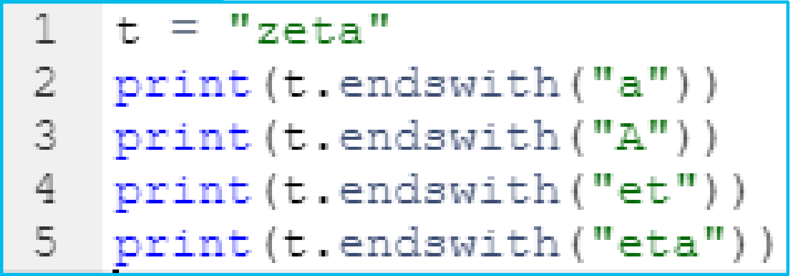


```
# Demonstrating the endswith() method:
if "epsilon".endswith("on"):
    print("yes")
else:
    print("no")
```

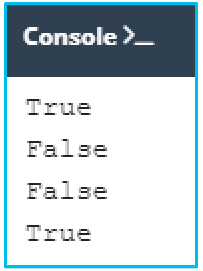


Console >_

```
yes
```

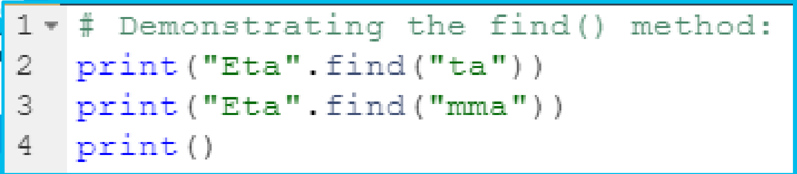


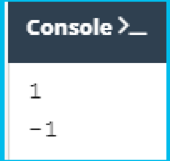
```
t = "zeta"
print(t.endswith("a"))
print(t.endswith("A"))
print(t.endswith("et"))
print(t.endswith("eta"))
```



Console >_

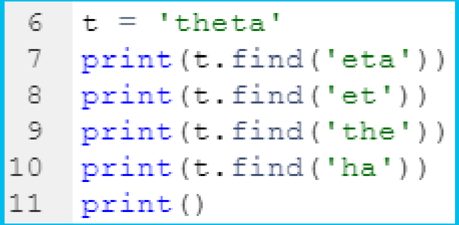
```
True
False
False
True
```

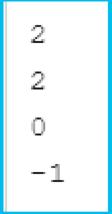

```
# Demonstrating the find() method:
print("Eta".find("ta"))
print("Eta".find("mma"))
print()
```



```
1
-1
```

The **find()** method


```
t = 'theta'
print(t.find('eta'))
print(t.find('et'))
print(t.find('the'))
print(t.find('ha'))
print()
```

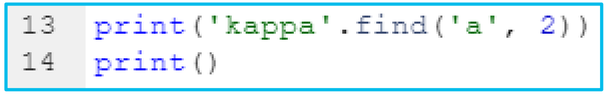

```
2
2
0
-1
```

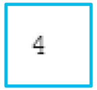
The **find()** method is similar to index(), which you already know - it looks for a substring and **returns the index of first occurrence of this substring**, but:

- **it's safer** - it doesn't generate an error for an argument containing a non-existent substring (it returns -1 then)
- **it works with strings only** - don't try to apply it to any other sequence.

Note: **don't use find()** if you only want to check if a single character occurs within a string - the **in** operator **will be significantly faster**.

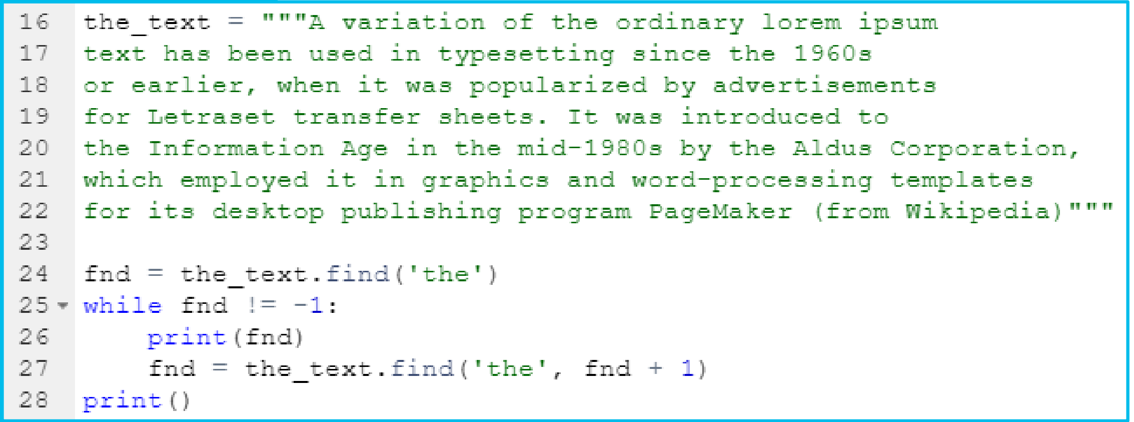
If you want to perform the find, not from the string's beginning, but from any position, you can use a two-parameter variant of the find() method. Look at the example:


```
print('kappa'.find('a', 2))
print()
```

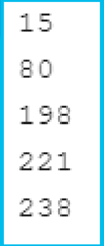

```
4
```

The **find()** method

You can use the find() method to search for all the substring's occurrences, like here:



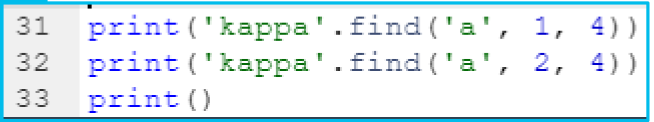
```python
the_text = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
or earlier, when it was popularized by advertisements
for Letraset transfer sheets. It was introduced to
the Information Age in the mid-1980s by the Aldus Corporation,
which employed it in graphics and word-processing templates
for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = the_text.find('the')
while fnd != -1:
    print(fnd)
    fnd = the_text.find('the', fnd + 1)
print()
```



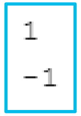
```
15
80
198
221
238
```

There is also a **three-parameter mutation** of the find() method - the third argument points to the first index which won't be taken into consideration during the search (it's actually the upper limit of the search).

The second argument specifies the index at which the search will be started (it doesn't have to fit inside the string).

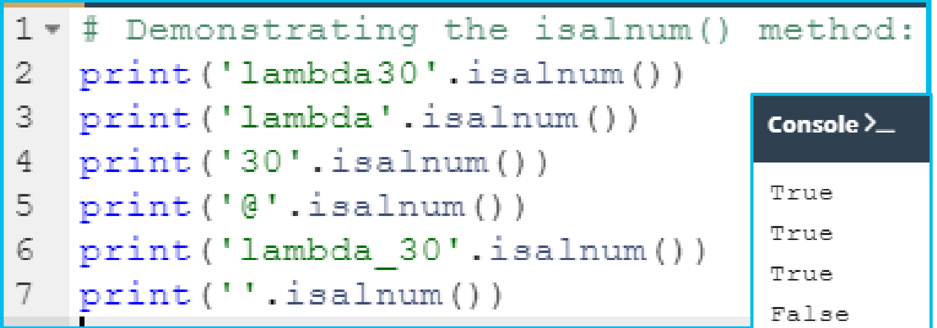


```python
print('kappa'.find('a', 1, 4))
print('kappa'.find('a', 2, 4))
print()
```

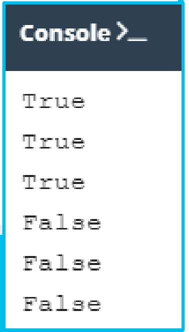


```
1
-1
```

(**a** cannot be found within the given search boundaries in the second print().



```
# Demonstrating the isalnum() method:
print('lambda30'.isalnum())
print('lambda'.isalnum())
print('30'.isalnum())
print('@'.isalnum())
print('lambda_30'.isalnum())
print(''.isalnum())
```

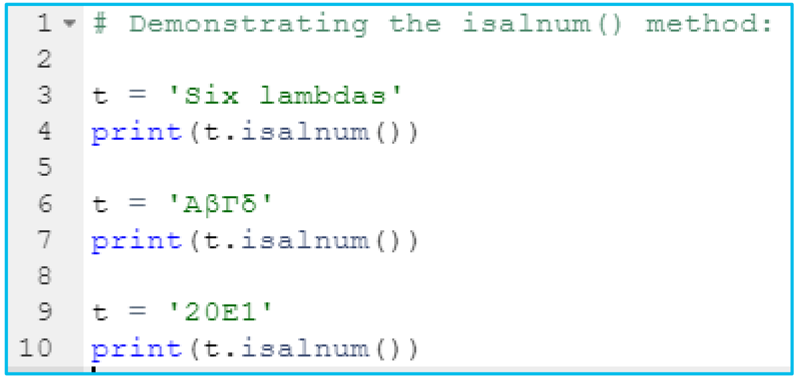


```
Console >_
True
True
True
False
False
False
```

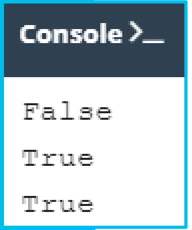
The **isalnum()** method

The parameterless method named **isalnum()** checks if the string contains only digits or alphabetical characters (letters), and returns True or False according to the result.

Note: any string element that is not a digit or a **letter causes the method to return False**. An **empty string does, too**.



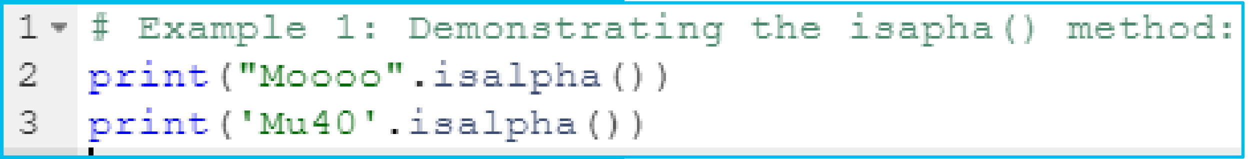
```
# Demonstrating the isalnum() method:

t = 'Six lambdas'
print(t.isalnum())

t = 'AβΓδ'
print(t.isalnum())

t = '20E1'
print(t.isalnum())
```



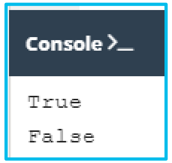
```
Console >_
False
True
True
```

The **isalpha()** method
The **isdigit()** method

The **isalpha()** method is more specialized - it's interested in letters only.

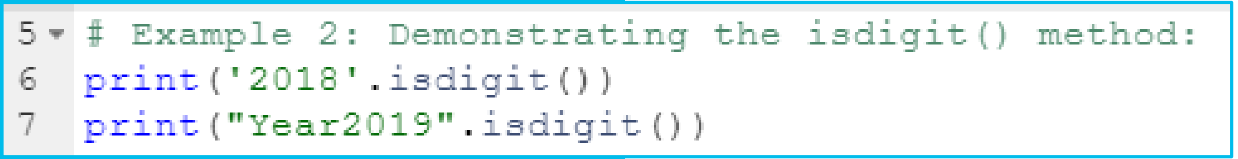


```
# Example 1: Demonstrating the isapha() method:
print("Moooo".isalpha())
print('Mu40'.isalpha())
```

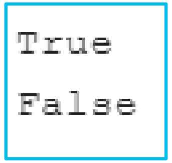


```
Console >_
True
False
```

In turn, the **isdigit()** method looks at digits only - anything else produces False as the result.



```
# Example 2: Demonstrating the isdigit() method:
print('2018'.isdigit())
print("Year2019".isdigit())
```



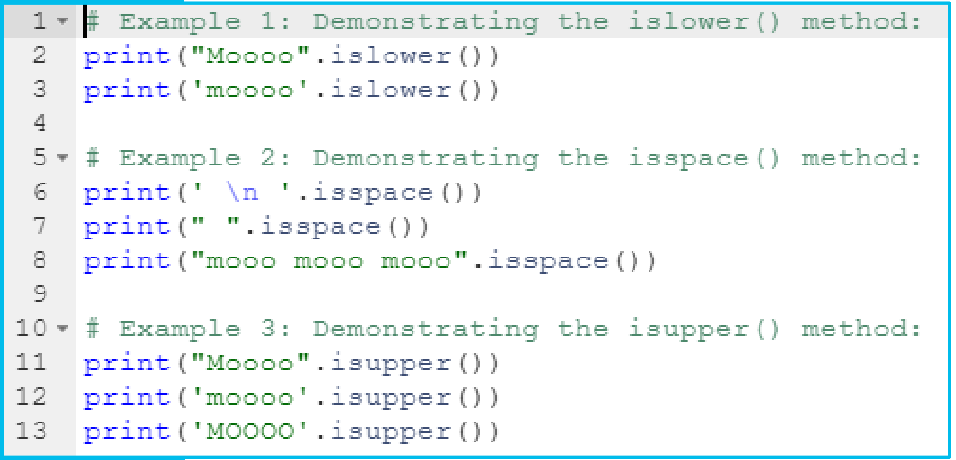
```
True
False
```

The **islower()** method *The **isspace()** method The **isupper()** method

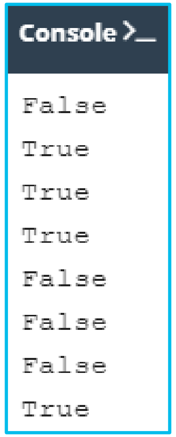
The **islower()** method is a fussy variant of isalpha() - it accepts lower-case letters only.

The **isspace()** method identifies whitespaces only - it disregards any other character (the result is False then).

The **isupper()** method is the upper-case version of islower() - it concentrates on upper-case letters only.



```python
# Example 1: Demonstrating the islower() method:
print("Moooo".islower())
print('moooo'.islower())

# Example 2: Demonstrating the isspace() method:
print(' \n '.isspace())
print(" ".isspace())
print("mooo mooo mooo".isspace())

# Example 3: Demonstrating the isupper() method:
print("Moooo".isupper())
print('moooo'.isupper())
print('MOOOO'.isupper())
```

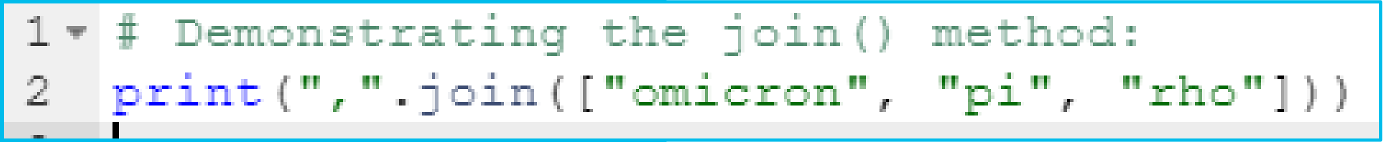


```
False
True
True
True
False
False
False
True
```

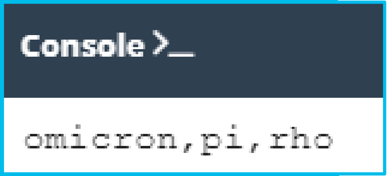
The **join()** method

The **join()** method is rather complicated, so let us guide you step by step thorough it:

- as its name suggests, the method performs a join - it expects one argument as a list; it must be assured that all the list's elements are strings - the method will raise a **TypeError** exception otherwise;
- all the list's elements will be joined into one string but...
- ...the string from which the method has been invoked is used as a separator, put among the strings;
- the newly created string is returned as a result.



```python
# Demonstrating the join() method:
print(",".join(["omicron", "pi", "rho"]))
```



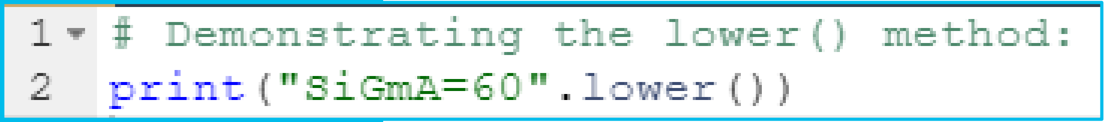
```
omicron,pi,rho
```

The **lower()** method

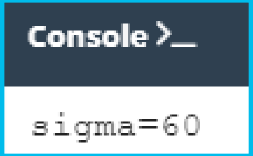
The **lower()** method makes a **copy of a source string**, **replaces all upper-case letters with their lower-case counterparts**, and **returns the string as the result.** Again, the source string remains untouched.

If the **string doesn't contain any upper-case characters**, the method **returns the original string**.

Note: The lower() method doesn't take any parameters.



```python
# Demonstrating the lower() method:
print("SiGmA=60".lower())
```

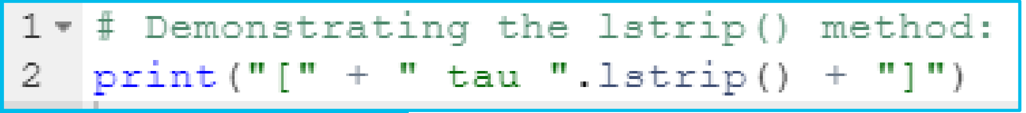


Console >_

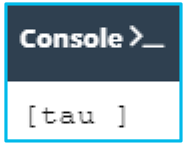
```
sigma=60
```

The **lstrip()** method LEFT

The parameterless **lstrip()** method **returns a newly created string formed from the original** one **by removing all leading whitespaces**.

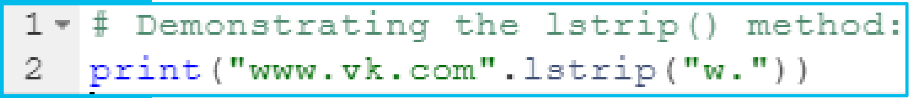


```python
# Demonstrating the lstrip() method:
print("[" + " tau ".lstrip() + "]")
```

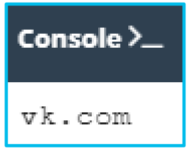


```
[tau ]
```

The **one-parameter lstrip()** method does the same as its parameterless version, but removes all characters enlisted in its argument (a string), not just whitespaces:

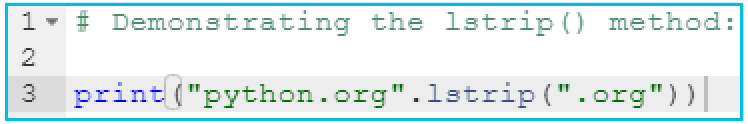


```python
# Demonstrating the lstrip() method:
print("www.vk.com".lstrip("w."))
```



```
vk.com
```

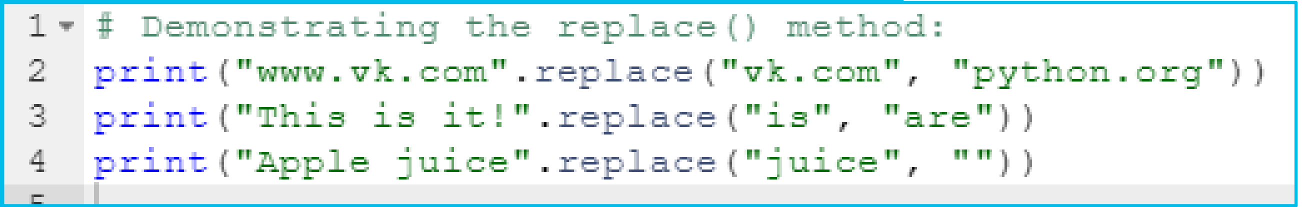
??????/

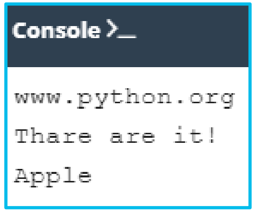


```python
# Demonstrating the lstrip() method:

print("python.org".lstrip(".org"))
```

The **replace()** method

The **two-parameter replace()** method returns a copy of the original string in which all occurrences of the first argument have been replaced by the second

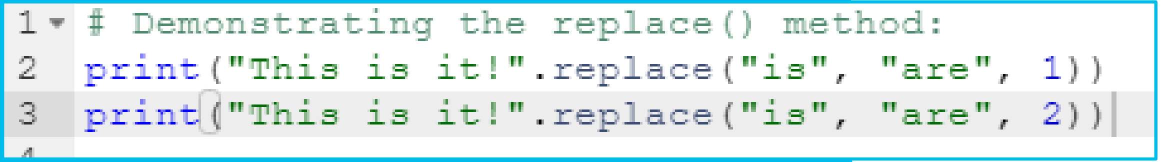

```python
# Demonstrating the replace() method:
print("www.vk.com".replace("vk.com", "python.org"))
print("This is it!".replace("is", "are"))
print("Apple juice".replace("juice", ""))
```

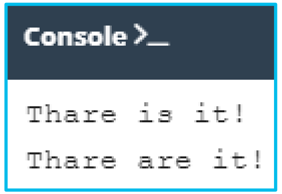


```
www.python.org
Thare are it!
Apple
```

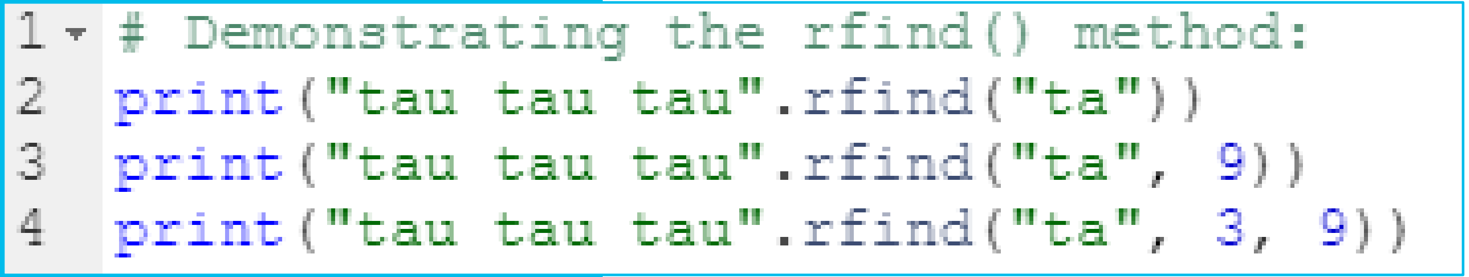
If the **second argument is an empty string**, replacing is actually removing the first argument's string. What kind of magic happens if the first argument is an empty string?

The **three-parameter replace()** variant uses the third argument (a number) to limit the number of replacements.

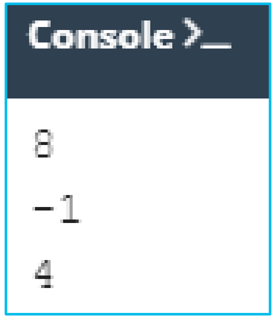

```python
# Demonstrating the replace() method:
print("This is it!".replace("is", "are", 1))
print("This is it!".replace("is", "are", 2))
```



```
Thare is it!
Thare are it!
```

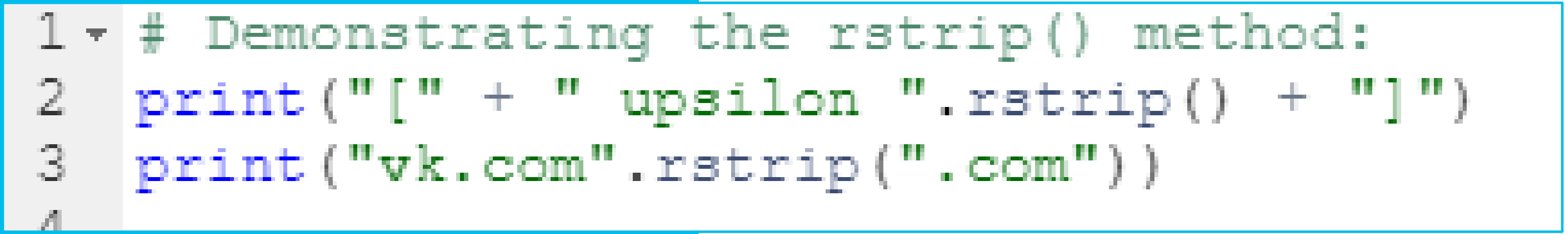


```python
# Demonstrating the rfind() method:
print("tau tau tau".rfind("ta"))
print("tau tau tau".rfind("ta", 9))
print("tau tau tau".rfind("ta", 3, 9))
```



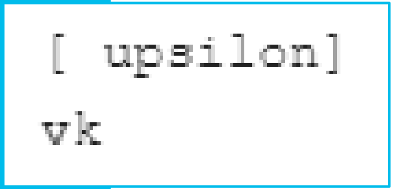
```
8
-1
4
```

The **rfind()** method

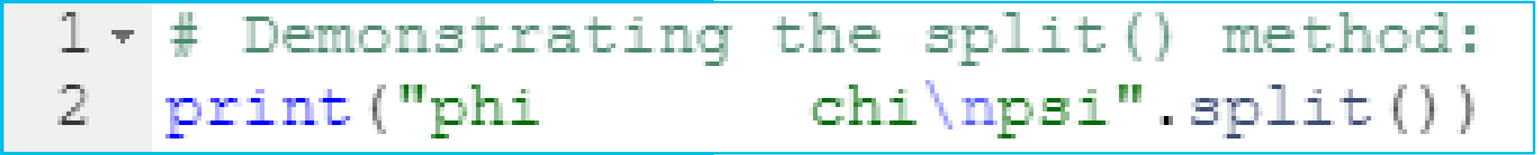


```python
# Demonstrating the rstrip() method:
print("[" + " upsilon ".rstrip() + "]")
print("vk.com".rstrip(".com"))
```

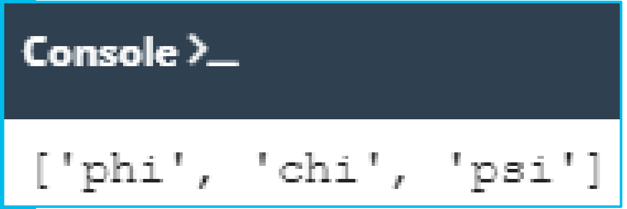
The **rstrip()** method



```
[ upsilon]
vk
```

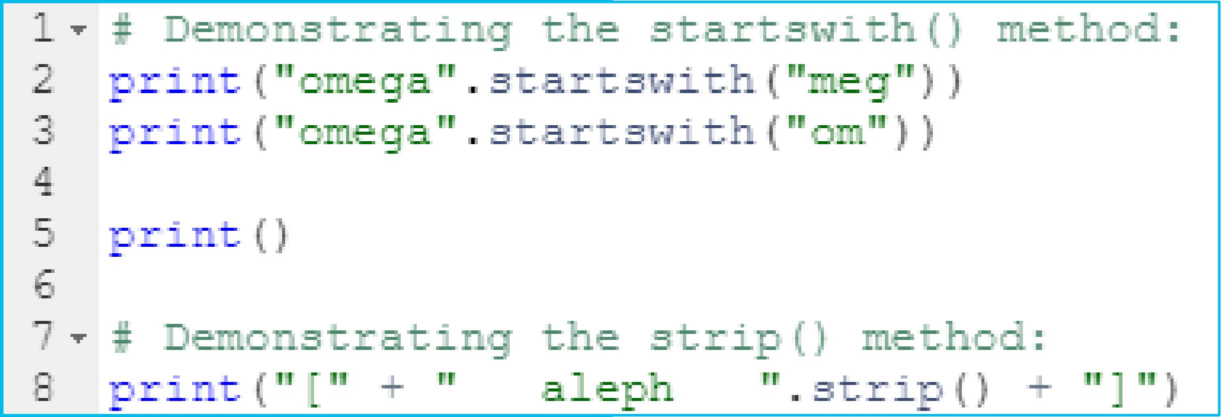


```
# Demonstrating the split() method:
print("phi             chi\npsi".split())
```



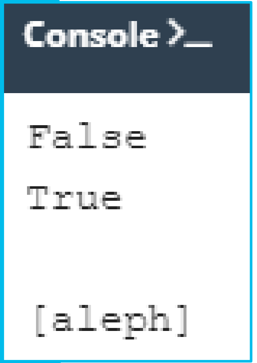
```
['phi', 'chi', 'psi']
```

The **split()** method



```
# Demonstrating the startswith() method:
print("omega".startswith("meg"))
print("omega".startswith("om"))

print()

# Demonstrating the strip() method:
print("[" + "    aleph    ".strip() + "]")
```

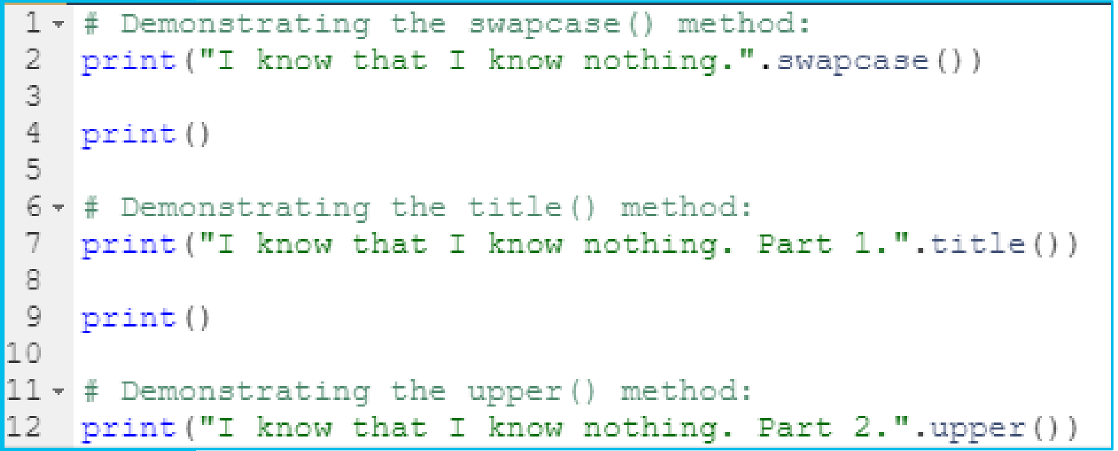
The **startswith()** and **strip()** methods



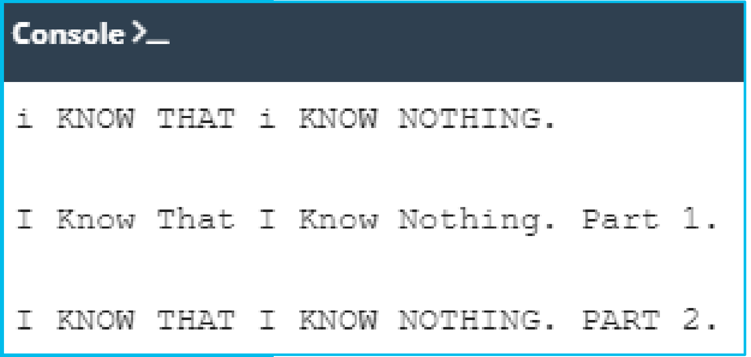
Console >_

```
False
True


[aleph]
```

The **swapcase(), title(), upper()** methods



```
# Demonstrating the swapcase() method:
print("I know that I know nothing.".swapcase())

print()

# Demonstrating the title() method:
print("I know that I know nothing. Part 1.".title())

print()

# Demonstrating the upper() method:
print("I know that I know nothing. Part 2.".upper())
```



Console >_

```
i KNOW THAT i KNOW NOTHING.

I Know That I Know Nothing. Part 1.

I KNOW THAT I KNOW NOTHING. PART 2.
```

Key takeaways

1. Some of the methods offered by strings are:

- capitalize() – changes all string letters to capitals;
- center() – centers the string inside the field of a known length;
- count() – counts the occurrences of a given character;
- join() – joins all items of a tuple/list into one string;
- lower() – converts all the string's letters into lower-case letters;
- lstrip() – removes the white characters from the beginning of the string;
- replace() – replaces a given substring with another;
- rfind() – finds a substring starting from the end of the string;
- rstrip() – removes the trailing white spaces from the end of the string;
- split() – splits the string into a substring using a given delimiter;
- strip() – removes the leading and trailing white spaces;
- swapcase() – swaps the letters' cases (lower to upper and vice versa)
- title() – makes the first letter in each word upper-case;
- upper() – converts all the string's letter into upper-case letters.

https://www.w3schools.com/python/python_ref_string.asp

Key takeaways

2. String content can be determined using the following methods (all of them return Boolean values):

- endswith() – does the string end with a given substring?
- isalnum() – does the string consist only of letters and digits?
- isalpha() – does the string consist only of letters?
- islower() – does the string consists only of lower-case letters?
- **isspace()** – does the string consists only of white spaces?
- isupper() – does the string consists only of upper-case letters?
- startswith() – does the string begin with a given substring?

https://docs.python.org/3/tutorial/datastructures.html

Examples

```
for ch in "abc123XYX":
    if ch.isupper():
        print(ch.lower(), end='')
    elif ch.islower():
        print(ch.upper(), end='')
    else:
        print(ch, end='')
```

```
s1 = 'Where are the snows of yesteryear?'
s2 = s1.split()
print(s2[-2])
```

```
the_list = ['Where', 'are', 'the', 'snows?']
s = '*'.join(the_list)
print(s)
```

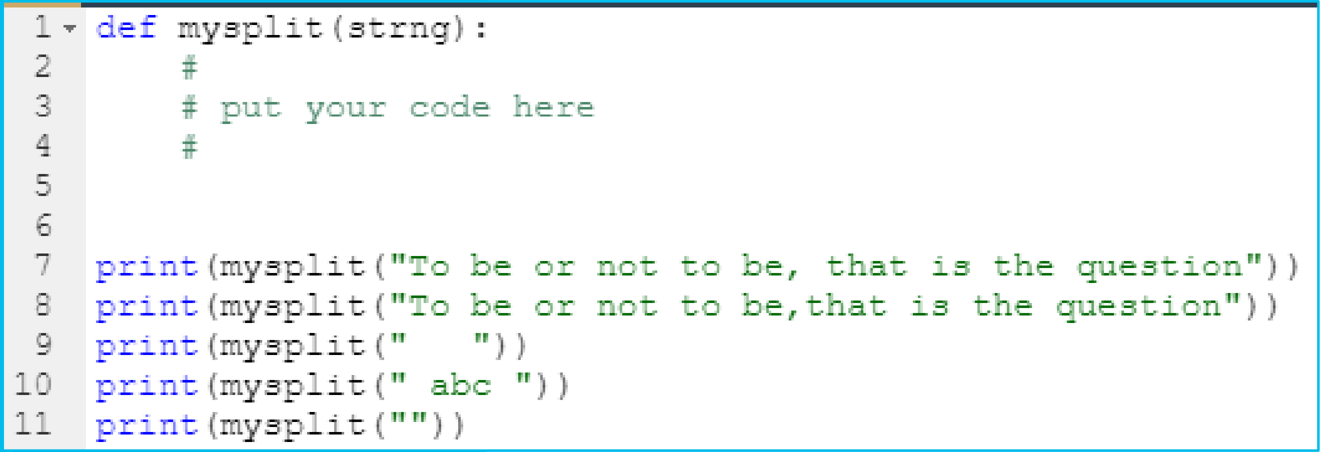
```
s = 'It is either easy or impossible'
s = s.replace('easy', 'hard').replace('im', '')
print(s)
```

Home work 8_1
"ff gg   hhhh"
['ff', 'gg', 'hhhh']

You already know how split() works. Now we want you to prove it.

Your task is to write your own function, which behaves almost exactly like the original split() method, i.e.:

- it should accept exactly one argument - a string;
- it should return a list of words created from the string, divided in the places where the string contains whitespaces;
- if the string is empty, the function should return an empty list;
- its name should be **mysplit()**
- Use the template in the editor. Test your code carefully.

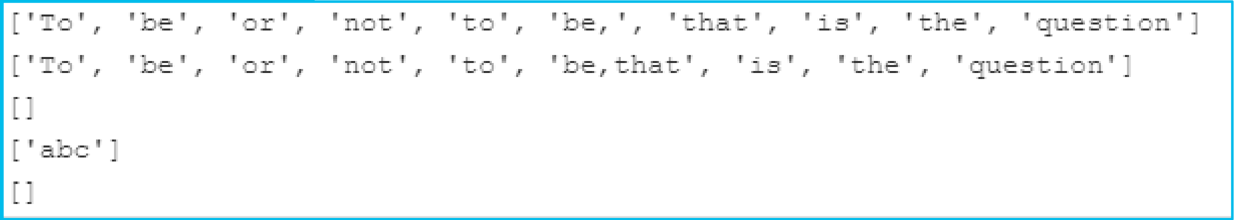


```python
def mysplit(strng):
    #
    # put your code here
    #


print(mysplit("To be or not to be, that is the question"))
print(mysplit("To be or not to be,that is the question"))
print(mysplit("   "))
print(mysplit(" abc "))
print(mysplit(""))
```

Home work 8_1 main.py

"To be or not to be, that is the question"
"To be or not to be,that is the question"
"   "
" abc ")
""

If …. == '__main__'



```
['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']
['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']
[]
['abc']
[]
```

Home work 8_1 USE PACKAGE

package mysplit
modules inside

use structure like:

```
Vasia_Pupkin_mysplit_8_1/|
                         |- modules/
                         |-packages/|
                         |          |
                         |          |-mysplit/
                         |                |__init__.py
                         |                |-mysplit_module.py
                         |-main.py
```

ЗАДАНИЯ

1) Прорешать всю классную работу
2) Выполнить все домашние задания

Почитать:

1) Byte of Python

**) Structuring Your Project:

Крайний срок сдачи 12/10 в 21:00 (можно раньше, но не позже)

https://docs.python-guide.org/writing/structure/

ЗАДАНИЯ

Название файлов, которые вы отправляете мне в telegram:

Vasia_Pupkin_class_work_L8_1.py,
Vasia_Pupkin_mysplit_8_1.zip/rar

Формат сообщения которое вы присылаете мне
(после полного выполнения домашнего задания, только один раз) в Telegram:
Добрый день/вечер. Я Вася Пупкин, и это мои домашние задания к лекции 8 часть про строки.
И отправляете файлы

Крайний срок сдачи 12/10 в 21:00 (можно раньше, но не позже)

https://docs.github.com/articles/using-pull-requests

Tap to links if you want to know more

Work with files:
https://www.youtube.com/watch?v=oRr_bEXJbV0
https://www.w3schools.com/python/python_ref_file.asp

Books for great peoples:
992 pages of "real" python

993 pages of "real" python

Watch this channel, useful things:
https://www.youtube.com/c/egoroffchannel/playlists

https://www.w3schools.com/python/default.asp

https://www.youtube.com/channel/UCr-KbmZWfDyTbqT_clZmhfw/videos

Q&A

Create your possibilities. Bye bye.