STRINGS

STRINGS

- Strings are immutable!
  - That means that a string cannot be changed once it has been created!
  - This is very important!

STRINGS

- We‘ve seen the string data type but so far we´ve only seen it used as a variables that can contain words
  - You can think of strings as a sequence of characters / symbols
  - As long as the characters/symbols are between matching quotes the characters/symbols are a valid string
    - Do note that some characters are not visible
      - Even though they are not visible, they are perfectly valid characters
      - An example of an invisible character would be the tab character (\t) or the new line character(\n)

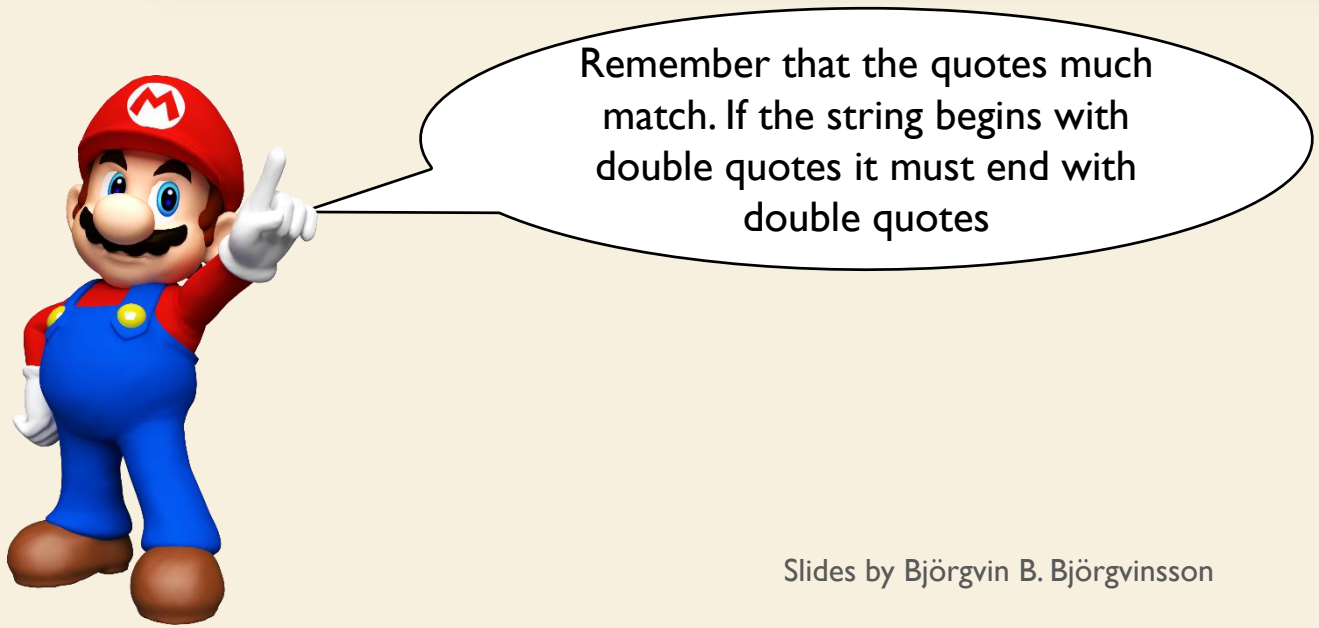
The quotes must match! If you start your string with double quotes you must end it with double quotes! The same goes for single quotes.

```
my_favorite_food = "pizza"
my_other_favorite_food = 'double cheeseburger from Burger King'
```

STRINGS

- We've already learned how to create strings but here is a quick refresher

```
my_favorite_food = "pizza"
my_other_favorite_food = 'double cheeseburger from Burger King'
```



STRINGS

- It is worth mentioning that a string that contains no characters is called the empty string and it is a valid string!

This is a string instance/variable that is has the name empty_string and has the value of the empty string (a set of double quotes with nothing between them). It is also worth mentioning that the empty string has a length of 0

```
1  empty_string = ""
2
```

STRINGS

- But what if I wan‘t to show quotes in my string?
  - There are multiple ways to do it!
  - If you want to use double quotes within your string you could surround your string with single quotes
  - If you want to use single quotes within your string you could surround your string with double quotes
  - You could escape the quote character by putting a forward slash(\) in front of it

```
1  a = "that's"
2  b = 'that\'s'
3  c = 'He said "good morning"'
4  d = "He said \"good morning\""
5
6  print(a) # print that's
7  print(b) # print that's
8  print(c) # prints He said "good morning"
9  print(d) # prints He said "good morning"
```

STRINGS

- String Representation
  - Every character is "mapped" (associated) with an integer
  - UTF-8, subset of Unicode, is such a mapping
  - The function `ord()` takes a character and returns its UTF-8 integer value, `chr()` takes an integer and returns the UTF-8 character.

```
1   print(ord('a')) # prints: 97
2
3   print((chr(104))) # prints: 'h'
```

STRINGS

- Here is a subset of the utf-8 mapping
  - You can se here that the character 'A' maps to the integer 65
  - The character '+' maps to the integer 43
  - And so on ...

Char	Dec	Char	Dec	Char	Dec
SP	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116

STRINGS

- Because a string is a sequence, we can associate each element of the string with an ***index***, a location within the sequence:
  - The first character is at index(position) 0
  - Python is quite clever and uses the negative numbers to access elements of the sequence from the end of the string

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

STRINGS

- A particular element of the string is accessed by the index of the element surrounded by square

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

Referencing the character at index 0

Referencing the character at index 4

Referencing the character at index -1

```
1   characters = "Hello world"
2
3   print(characters[0]) # prints: 'H'
4
5   print(characters[4]) # prints: 'o'
6
7   print(characters[-1]) # prints: 'd'
```

STRINGS

- But we have be careful not to reference an index that is out of range
  - In this example the highest index has a value is index 10(the character d) and the lowest is -11 (the character H)

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

```
1   characters = "Hello world"
2
3   print(characters[-11]) # prints: 'H'
4
5   print(characters[34]) # produces an error
```

This will raise an index out of range error/exception

STRINGS

- Since strings are a sequence we can use strings with loops

This loops will iterate over the string that is stored in the variable called name. The varialbe letter will take the value of each character in the string and print it to the screen, one character per line

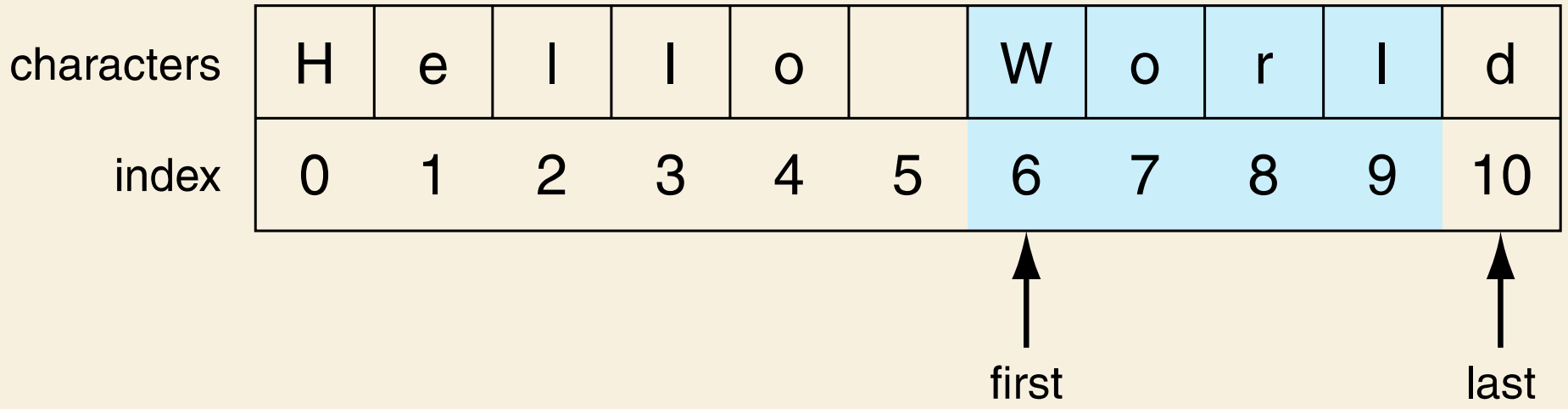
```
1    name = "Bruce Wayne"
2
3    for letter in name:
4        print(letter)
```

STRINGS

- String slicing
  - String slicing creates a new string! **Remember, string are immutable**!
  - Is the ability to select a subsequence of the overall sequence
  - In other words, it means you can get a part of the string
  - Slicing uses the syntax [start : finish]
    - start is the index of where we start the subsequence
    - finish is the index of where we end the subsequence,
      - Do note that the value of finish is not part of the subsequence
  - If start or finish are not provided, start will default to the beginning of the sequence and finish will the will default to the end of the sequence

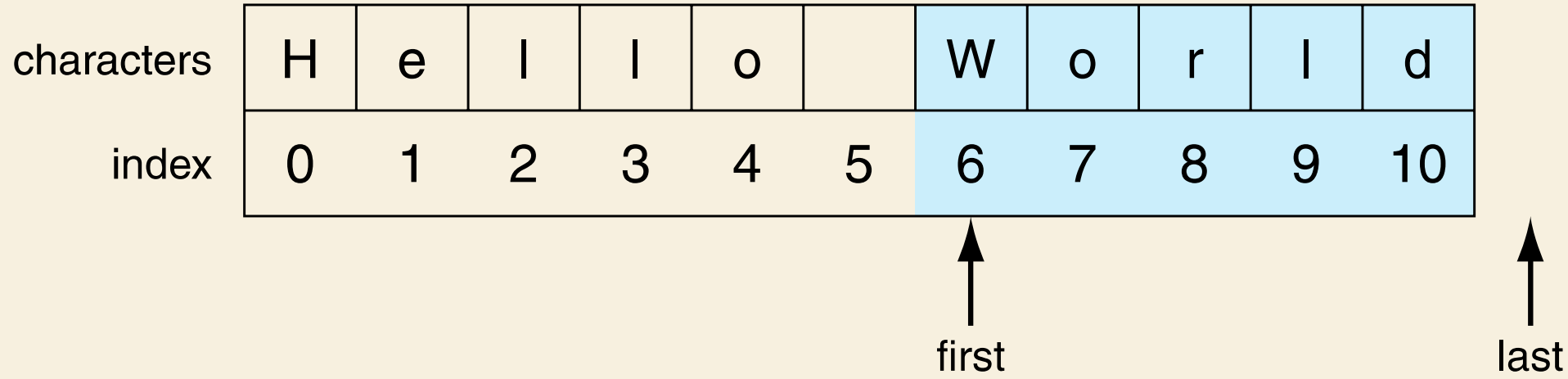
STRINGS

```
helloString[6:10]
```



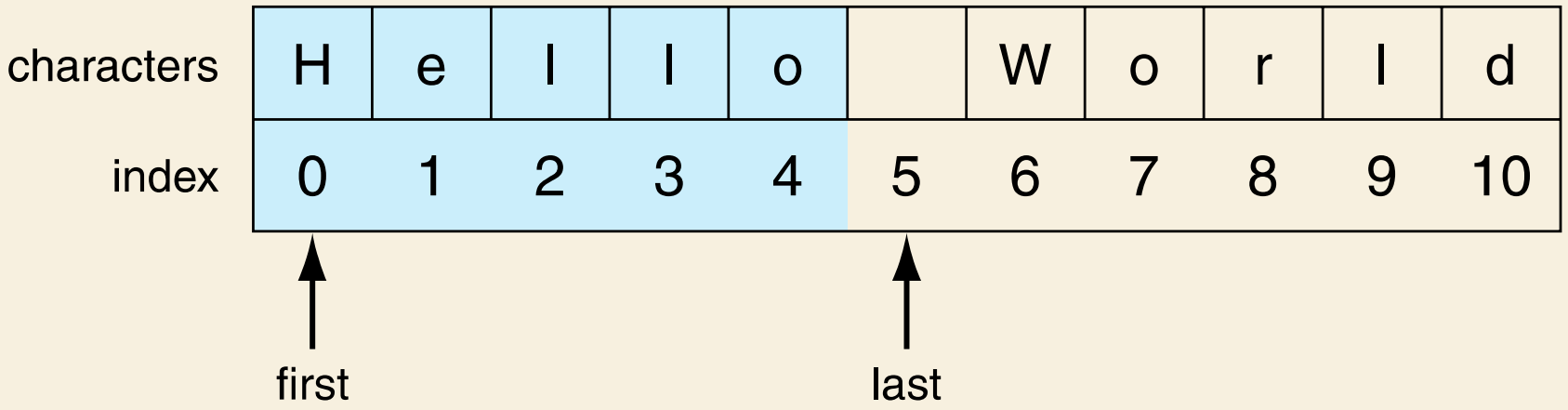
STRINGS

```
helloString[6:]
```



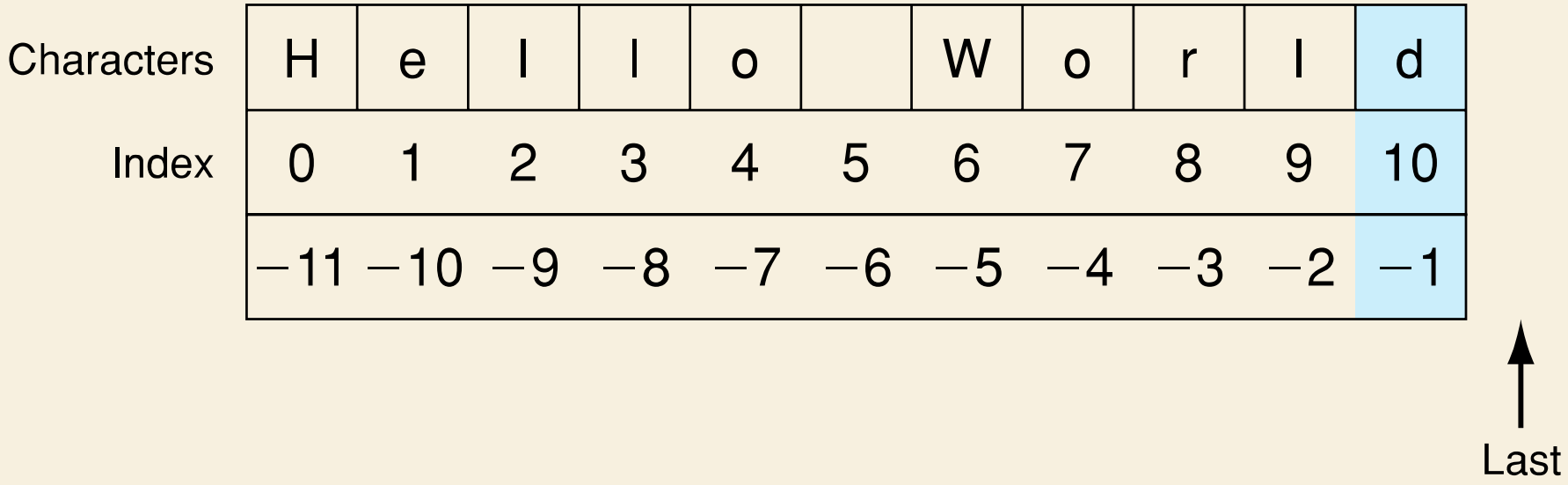
STRINGS

```
helloString[:5]
```



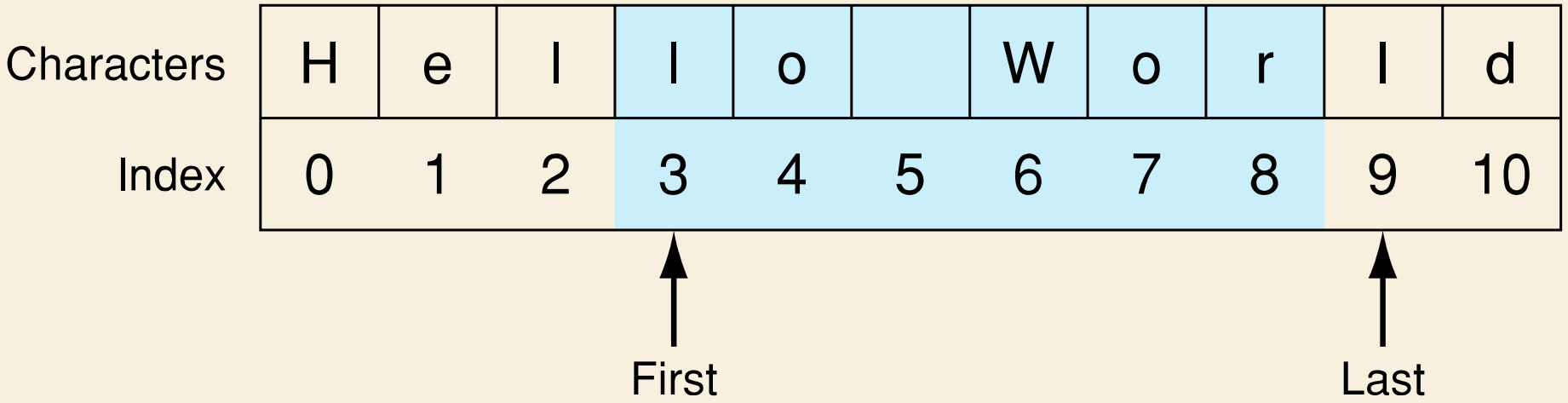
STRINGS

```
helloString[-1]
```



STRINGS

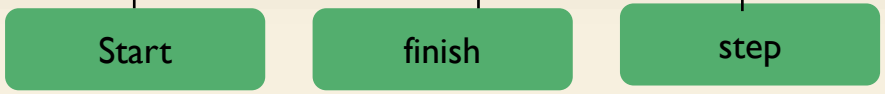
```
helloString[3:-2]
```



STRINGS

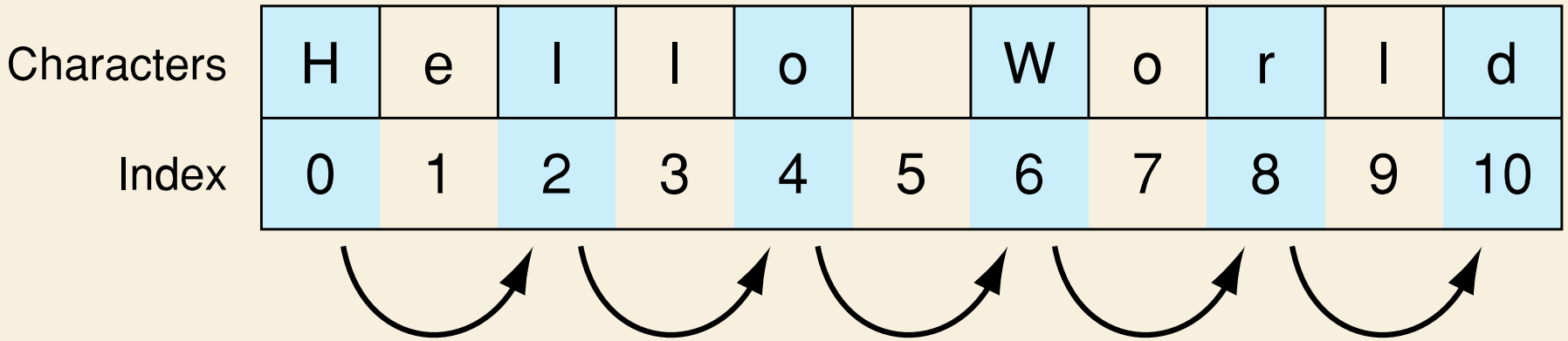
- We can add the third value between when slicing strings
  - The third value is an indicator of how big a step to take

```
1   characters = "Hello world"
2
3   new_str = characters[0:11:2] # creates the string 'Hlowrd'
```



STRINGS

```
helloString[::2]
```



STRINGS

- Here are two Python tricks
  - Do note that these tricks are very specific for Python

```
1   my_str = 'hi mom'
2   new_str = my_str[:] # creates a copy of my_str
3
4   my_str = "madam I'm adam"
5   reverseStr = my_str[::-1] # creates a reversed copy of my_str
```

COMMON STRING OPERATIONS AND FUNCTIONS

STRING OPERATIONS

- We can use some math operands with strings
  - (+) means concatenation

```
first_name = "john"
last_name = "doe"
full_name = first_name + " " + last_name

print(full_name) # prints: 'john doe'
```

  - (*) repeats the string

```
word = "nana"
word_times_three = word * 3
print(word_times_three) # prints: 'nanananananana'
```

STRING OPERATIONS

- Both + and * on strings create new a string, does not modify the arguments
  - Remember that strings are immutable
- Order of operations is important for concatenation, irrelevant for repetition
- The types required are specific
  - For concatenation you need two strings
  - For repetition a string and an integer

STRING OPERATIONS

- the + and * operators is ***overloaded***
  - This means that what the + and * operation perform depends on the types it is working on

the + performs concatenation because the operands are strings

the + performs addition because the operands are ints

```
1  first_name = "john"
2  last_name = "doe"
3
4  full_name = first_name + " " + last_name
5  sum_of_two = 2 + 4
```

*the * performs repetition because the operands are a string and an int*

*the * performs multiplication because the operands are ints*

```
1  word = "nana"
2  long_word = word * 3
3
4  product_of_two = 2 * 2
```

STRING OPERATIONS

- Since strings are a sequence of characters it makes sense that we can check if some character is in the sequence
  - Python makes this easy for us with the **in** operator

```
1   greeting = "hello world"
2
3   if 'h' in greeting:
4       print("the letter h is in the variable!")
```

The in operator returns True if the character on the left hand side is present in the string. It returns False otherwise.

STRING OPERATIONS

- len() is a function commonly used with strings. Note that it can also be used with other sequence types
  - It returns the length of the sequence
- Do note that the last character of a string is always on index -1 and on index len(variable_name)-1
  - That means that if the length of the string is 8 (the string contains 8 characters) the index of the last character is 8 - 1

```
1   greeting = "Hello!"
2
3   length = len(greeting) # returns the number 6
```

STRING OPERATIONS

- The str() function can be used to create a string representation of a variable

```
number = 8
number_str = str(number) # creates the string '8'

other_number = 3.14
other_number_str = str(other_number) # creates the string '3.14'
```

STRING METHODS

STRING METHODS

- strings are objects and object can store values and have methods(functions) that can do something with those values
  - A method is just a function associated with an object
  - We will learn more about objects when we cover classes
- Every string we create has a set of methods that can be used on that string
  - We use dot notation to access these methods

```
1   name = "John"
2
3   upper_name = name.upper() # creates this string: "JOHN"
4   lower_name = name.lower()  # creates this string: "john"
```

STRING METHODS

- It is called dot notation when we add a dot after the variable name followed by the method name

```
1   name = "John"
2
3   upper_name = name.upper() # creates this string: "JOHN"
4   lower_name = name.lower()  # creates this string: "john"
```

We use the dot after the variable name to access the method on that particular variable

STRING METHODS

- **upper()** and **lower()** are string methods that are commonly used
  - **lower()** creates a new string with only lowercase letter
  - **upper()** creates a new string with only uppercase letter
  - Remember! Strings are immutable the string in the variable name will not change!

```
1   name = "John"
2
3   upper_name = name.upper() # creates this string: "JOHN"
4   lower_name = name.lower()  # creates this string: "john"
```

We use the dot after the variable name to access the method on that particular variable

STRING METHODS

- replace() is a method that creates a new string with some replacements

```
name = "John"

new_name = name.replace('h', 'a') # creates anew string where all occurences of 'h' have been replaced by 'a'

print(name) # prints: 'John'
print(new_name) # print: 'Joan'
```

STRING METHODS

- We can use some methods to ask questions about the string such as:

```
characters = "Hello"

characters.isdigit() # this methods returns True if the characters can be interpreted as a number, False otherwise
characters.isalpha() # this methods returns True if all the characters are alphabetical characters, False otherwise
```

Returns True if all characters of the variable are digits

Returns True if all characters of the variable are alphabetic characters

```
characters = "Hello"

if characters.isdigit() :
    print('The variables characters only contains digits')
elif characters.isalpha():
    print('The variables characters only contains alphabetic characters')
```

STRING METHODS

- The strip method can come in very handy
  - It creates a new string with no leading or trailing whitespaces
  - It also deletes trailing newline characters
  - This is commonly used to clean up user input

```
1  greeting = '    hello world     \n'
2  stripped_greeting = greeting.strip() # deletes all leading and trailing whitespaces from the string
3  |   |   |   |   |   |   |   |   # also deletes trailing newline characters
4  print(greeting) # prints: '    hello world     '
5  print(stripped_greeting) # prints: 'hello world'
```

STRING METHODS

- We can chain methods
  - That means that after one method we can simply call another one

```
1   user_input = input("Enter somthing: ")
2
3   clean_user_input = user_input.strip().lower()
4
5   print(clean_user_input)
```

First we call the strip method. The strip methods returns a string with no leading or trailing whitespaces. Next we call lower() on the string returned by strip(). That string is then stored in the variable clean_user_input

STRING METHODS

- The split() method is also used very much

```
greeting = 'hello world'
greeting_list = greeting.split() # creates the list [hello, word]
# if no argument is provided the split method will split the string on a space
```

```
greeting = "hell, no, world"
greeting_list = greeting.split(',') # here a seperator value is provided so the string will be split using that seperator
                                    # the method will return the list [hell, no, world]
```

STRING METHODS

- Here is a common use case for the split method

```
1   words = input("Enter 3 words seperated by a comma: ")
2   words = words.split(',')
3
4   for word in words:
5       print(word.strip())
```

STRING METHODS

- Here is a complete list of the member functions of the string class along with the operators that have been overloaded for the string class

capitalize( )	lstrip([*chars*])
center(*width*[, *fillchar*])	partition(*sep*)
count(*sub*[, *start*[, *end*]])	replace(*old*, *new*[, *count*])
decode([*encoding*[, *errors*]])	rfind(*sub* [,*start*[,*end*]])
encode([*encoding*[,*errors*]])	rindex(*sub*[, *start*[, *end*]])
endswith(*suffix*[, *start*[, *end*]])	rjust(*width*[, *fillchar*])
expandtabs([*tabsize*])	rpartition(*sep*)
find(*sub*[, *start*[, *end*]])	rsplit([*sep* [,*maxsplit*]])
index(*sub*[, *start*[, *end*]])	rstrip([*chars*])
isalnum( )	split([*sep* [,*maxsplit*]])
isalpha( )	splitlines([*keepends*])
isdigit( )	startswith(*prefix*[, *start*[, *end*]])
islower( )	strip([*chars*])
isspace( )	swapcase( )
istitle( )	title( )
isupper( )	translate(*table*[, *deletechars*])
join(*seq*)	upper( )
lower( )	zfill(*width*)
ljust(width[, *fillchar*])	