Ohjelmoinnin perusteet
Luento 3 – Strings

String data type

String is a textual data type that contains a sequence of characters. **Strings** are created and denoted with double or single **quotation marks** around the characters. There are no limitations for the type of characters.

```
>>> "string1"
'string1'
>>> 'string2'
'string2'
>>>
```

Three double or single quotation marks denote a **multiline string** that can cover multiple lines.

```
>>> x = """this is
a multiline
string"""
>>> print(x)
this is
a multiline
string
>>>
```

Indexing

Individual characters of strings are accessed with **square brackets** after the string and an **index**. Indexing starts from 0, so the first element of string x is accessed with syntax x[0]. Negative indices mean counting the characters from the end of the string. The element of x in the index len(x)-1 is always the last element.

```
>>> x = "string1"
>>> x[0]
's'
>>> x[1]
't'
>>> x[2]
'r'
>>> x[-1]
'1'
>>> x[-2]
'g'
>>> x[len(x)-1]
'1'
>>>
```

Slices of strings can be accessed with the **colon notation index n:m**, which includes all the elements from the index n to index m-1. So the last element is not included. Index n:n returns an empty string. If the index n is missing, the indexing starts from the beginning, and if m is missing, the indexing goest to the end.

```
b = "Hello, World!"
print(b[2:5]) # elements from index 2 to 4 (does not include 5)
print(b[:5]) # elements from index 0 to 4 (does not include 5)
print(b[2:]) # elements from index 2 to the last
print(b[:]) # all the elements
print(b[-5:-2]) # elements from index -5 to -3 (does not include -2)
print(b[:-1]) # elements from index 0 to -2 (does not include -1)
print(b[-4:]) # elements from index -4 to -1
# indices can also be applied straight into a string literal
print("Hello, World!"[2:5])
```

String operators

Strings have two **operators + and *.** The plus operator concatenates the given strings and the times operator concatenates the given string with the given number of times.

```
>>> a = "Hello"
>>> b = "world"
>>> a+b
'Helloworld'
>>> a*3
'HelloHelloHello'
>>> a + "" + b + "!"
'Helloworld!'
>>> a = "Hello"
>>> b = "world"
>>> a+b
'Helloworld'
>>> a*3
'HelloHelloHello'
>>> a + " " + b + "!"
'Hello world!'
>>>
```

Escape character

Special characters can be inserted into strings with **escape character \.** Below is a table of them.

Escape Sequence	Meaning
`\newline`	Backslash and newline ignored
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\ooo`	Character with octal value *ooo*
`\xhh`	Character with hex value *hh*

Escape Sequence	Meaning
`\N{name}`	Character named *name* in the Unicode database
`\uxxxx`	Character with 16-bit hex value *xxxx*
`\Uxxxxxxxx`	Character with 32-bit hex value *xxxxxxxx*

The characters \n will insert a new line and \t will insert a tabulator space. In case you want a string that is printed exactly with the characters they are containing, you can use **raw strings**, which are created with the prefix r before the string literal. This means the escape charaters are not interpreted as special characters.

```
>>> print("Hello\nWorld!")
Hello
World!
>>> print("Hello\tWorld!")
Hello   World!
>>> print(r"new line \n will not appear")
new line \n will not appear
>>>
```

Escape characters and raw strings are explained in more detail in Python documentation:
https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals

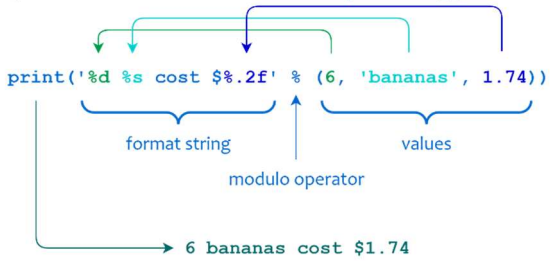
String formatting

Formatting strings = inserting strings inside other strings with some given formatting criteria.

Old formatting operator %. The % operator inside a string is replace with the object after the string inside the formatting operator % in the corresponding order. The % operators in the string need to have some conversion type: d=decimal integer, s=string.

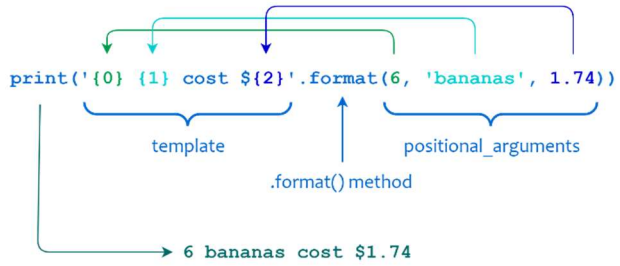
```
>>> "%s is %d years old." %("John", 23)
'John is 23 years old.'
>>>
```

Below is a picture that demonstrates the formatting operator %. You can read more about the formatting operator from the website: https://realpython.com/python-input-output/#the-string-modulo-operator



New formatting operator {}. The {} operator will be replaced with given arguments of format() method that is applied to the string. There can also be numbers inside the operators, which means the arguments are replaced in this order. If there are keywords, they will be replaced with the given keywords.

```
>>> txt = "I want {} pieces of item {} for {:.1f} dollars."
>>> print(txt.format(3, 567, 49.95))
I want 3 pieces of item 567 for 50.0 dollars.
>>> txt = "I want to pay {2} dollars for {0} pieces of item {1}."
>>> print(txt.format(3, 567, 49.95))
I want to pay 49.95 dollars for 3 pieces of item 567.
>>> txt = "I have a {carname}, it is a {model}."
>>> print(txt.format(carname = "Ford", model = "Mustang"))
I have a Ford, it is a Mustang.
>>>
```



Formatted string literals, f-strings (Python version 3.6+). The new formatting operator {} can also be used without the format() method if you put letter f before the string, which makes it so called f-string.

```
>>> a = 5
>>> b = 10
>>> print(f"Five plus ten is {a + b} and not {2 * (a + b)}.")
Five plus ten is 15 and not 30.
>>>
```

You can read more about the new formatting operator {} and f-strings from the website: https://realpython.com/python-formatted-output/

String methods

Method = function that can be applied to object with the dot notation: object.function()

Strings can be modified with a set of built-in methods. The methods always return a new string, because **string is immutable data type**. The methods are used with the **dot notation**: string.method()

```
>>> "Hello, World!".upper()
'HELLO, WORLD!'
>>> "Hello, World!".replace("H", "J")
'Jello, World!'
```

Below is a list of the most common string methods.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found

rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning