

Python Dersleri Notları

Input: print('being a good person')

Output: being a good person

Input: print('clarusway will change my life')

Output: clarusway will change my life

Tips:

- Surrounding the expression with triple quotes: `"""..."""` or `'...'` ensures that the code returns no error, especially in long texts.
- Use double quotes if your string includes the single one: e.g. "It's my pleasure!"
- Use single quotes if your string includes the double one: e.g. 'He said: "I am done" and fell down.'
- Use triple quotes if your string is too long which composed of multiple lines : e.g. :

"""

...long string...

..long string..

"

Input: print("it's not a problem")

Output: it's not a problem

Input: print("""it's not a problem using "triple" quotes""")

Output: it's not a problem using "triple" quotes

Tips:

- If you have noticed we used `“.”` not `“,”` for the decimal number: 3.14
- Surrounding the expression with quotes makes it in string type.

Input: print(3.14)

Output: 3.14

Input: print('3.14')

Output: 3.14

Input: print(2+2)

Output: 4

If you need an empty line, you can use only print() function.

Input:

```
print('first line')
```

```
print() # second line is empty
```

```
print('third line')
```

Output:

first line

third line

Format & Style of Coding

PEP 8:

PEP stands for Python Enhancement Proposal. PEP 8 is a coding convention, a set of recommendation, about how to write your Python code more readable. In other words, PEP 8 is a document that gives coding conventions for the Python code comprising the standard library in the main Python distribution.

Some Important PEP 8 Rules:

- Limit all lines to a maximum of 79 characters. For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
- Spaces are the preferred indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python 3 disallows mixing the use of tabs and spaces for indentation.
- Avoid extraneous whitespaces in the following situations:

-Immediately inside parentheses, brackets or braces :

YES : `spam(meat[1], {milk: 2})` , **NO :** `spam(meat[1], { milk: 2 })`

-Between a trailing comma and a following close parenthesis :

YES : `df[0,] or foo = (2,)` , **NO :** `df[0,] or foo = (2,)`

-Immediately before a comma, semicolon, or colon :

YES : `if y == 3: print x, y; x, y = y, x` , **NO :** `if y == 3 : print x , y ; x , y = y , x`

-Immediately before the open parenthesis that starts the argument list of a function call:

YES : `print('peace')` , **NO :** `print ('peace')`

-More than one space around an assignment (or other) operator to align it with another:

YES :

`x = 3`

`y = 4`

`long_vars = 5`

NO :

`x = 3`

`y = 4`

`long_vars = 5`

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker.
- Always surround these binary operators with a single space on either side: assignment (`=`), augmented assignment (`+=`, `-=`, etc.), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).

Comments: are used to explain code when the basic code itself isn't clear. Python ignores comments, and so will not execute code in there, or raise syntax errors for plain English sentences.

There are three types of commenting methods. These are :

-Single-line comments: begin with the hash character `#` and are terminated by the end of the line. `#` sign converts all subsequent characters to the comment form that Python does nothing.

-Inline comments: also begin with hash character `#` and start from the end of a code line.

Input: `print('the cosmos has no superiority to chaos') # This is an inline comment`

Output: the cosmos has no superiority to chaos

Multi-line comments basically consist of multiple comment lines.

Input:

`print(3 + 4)`

`# This is the multi-line comment, line-1`

`# This is the multi-line comment, line-2`

`# This is the multi-line comment, line-3`

Output: 7

Tips:

- To begin with, after `#` there should be one space, and in the inline comments, there should be at least two spaces between the end of the code and `#`.
- A comment is not a piece of code. It should be short. It's better to split a long comment into multiple lines. You have to add `#` at the beginning of each new line.

Docstrings: are - unlike regular comments - stored as an attribute of the function or the module they document, meaning that you can access them programmatically. Docstring runs as an explanatory text of codes and it should be written between triple quotes. Like: `"""docstring"""`. Normally, when we want to call docstring of a function or module to read, we will use `__doc__` (the keyword doc enclosed by double underscores) syntax. See the example below : Here is an example:

Input:

```
def function(): # Don't be confused, we use 'def()' to create a function.
    # You will see it in the next lessons.
    """
    Hi, I am the docstring of this code.
    If you need any information about this function or module, read me.
    It can help you understand how the module or function works.
    """
    print(function.__doc__)
```

Output:

Hi, I am the docstring of this code.
If you need any information about this function or module, read me.
It can help you understand how the module or function works.

Naming Variables:

- Choose lowercase words and use underscore to split the words:

```
price = 22, 44, 66
low_price = 12.00
```

Do not use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single-character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. If you want to use 'l', use 'L' instead.

Do not use specific Python keywords (name of a function or phrase) as a name, like sum, max, min, in, or, for, etc.

- Use a sensible name.

Yes: figures = 'this is better',

No: f = 'it is not meaningful'

- Don't choose too common names. Use a name to describe the meaning of the variable. However, try to limit it to no more than 3 words.
- If the word you intend to choose is long, try to find the most common and expected short form to make it easy to predict later.

Eg:

-Cleaned data → `cleaned_data`

-Indexes of the Clear Application Syntaxes → `clr_app_syntax`

-Customer Information of the Bank Accounts → `customer_bank_info`

- We can assign a value to multiple variables.
e.g.: `variable1 = variable2 = 'clarusway opens your path'`.
- We can also assign multiple values to multiple variables in sequence using commas.
e.g.: `variable1 = "clarusway", "opens", "your", "path"`

Introduction to Data Types

Strings:

Input:

```
text1 = "I have learned strings" # everything can be string if it's surrounded with double or single quotes
print(text1) # string type is called str
```

Output:

```
I have learned strings
```

Input: `print('632')` # this is also a string type

Output: 632

Numeric Types:

- Integers (int): are whole numbers (positive, negative or zero), including no decimal point. (71, -122, 0)
- Floats (float): stand for real numbers with a decimal point. (71.0, -33.03)
- Complexes: are written in the form, $x + yj$, where x is the real part and y is the imaginary part. (3.14j)

71 and 71.0 have the same numerical value. But they differ in terms of numeric type. The types of these numbers are int and float respectively.

Boolean:

Boolean types are called bool and their values are the two constant objects False and True. They are used to represent truth values (other values can also be considered false or true). In numeric contexts (for example, when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. Note that we write the first letter of True in uppercase. This is the rule of Python that we must write like this : True, False.

```
tv_open = True # it seems TV is on now
is_rainy = False # I love sunny weather
```

Type Conversion:

We can convert the types of data to each other if the type allows to be converted. There are some functions:

- `str()` function converts to string type
- `int()` function converts to signed integer type
- `float()` function converts to floating point type
- `type()` function prints the types of data

Input:

```
example1 = 'sometimes what you say is less important than how you say it'
print(type(example1)) # type(example1) function gives the same result as output: str
Output: <class 'str'>
```

Input: `example2 = '71'`
`print(type(example2))`
Output: <class 'str'>

Input: `example3 = 71`
`print(type(example3))`
Output: <class 'int'>

Input: `example4 = 71.0`
`print(type(example4))`
Output: <class 'float'>

Input: `example5 = 3.14j`
`print(type(example5))`
Output: <class 'complex'>

Input: `example6 = True`
`print(type(example6))`
Output: <class 'bool'>

converting between different types:

Input:

```
f = 3.14 # the type is float
s = str(f) # converting float to string
print(type(s))
```

Output: <class 'str'>

Input:

```
f = 3.14 # the type is float
i = int(f) # while converting a float value to an integer its decimal part is disregarded
print(i, '\n') # '\n' gives us an empty line
print(type(i))
```

Output:

3

<class 'int'>

Input:

```
i = 3
f = float(i)
print(f, '\n')
print(type(f))
```

Output:

3.0

<class 'float'>

Input:

```
x = 39
v = "11"
y = "2.5"
z = "I am at _"
```

```
print(x-int(v))
print(x-float(y))
print(z+str(x))
```

Output:

28

36.5

I am at _39

Input: str(3.14) Output: '3.14'

Input: int(3.14) Output: 3

Input: float(11) Output: 11.0

Input: float("11") Output: 11.0

Input: float("33.33") Output: 33.33

Input:

```
x = 39
v = "11"
y = "2.5"
z = "I am at _"
print(x-int(v))
print(x-float(y))
print(z+str(x))
```

Output:

28

36.5

I am at _39

Variables:

Variable is a location designated where a value can be stored and accessed later. Imagine a box where you store something. That's a variable.

Let's create a box (variable) in which contains basketball balls. Let's name it ball_box. It is also the name of the variable.

Creating, naming the variable and assigning a value to it happen simultaneously by this syntax : ball_box = 20 basketball balls

Input:

```
color = 'red' # str type variable
season = 'summer'
price = 250 # int type variable
pi = 3.14 # float type variable
color = 'blue' # You can always assign a new value to a created variable
price = 100 # value of 'price' is changed
season = 'winter'
```

```
print(color, price, season, sep=', ')
```

Output:

```
blue, 100, winter
```

Arithmetic Operations:

```
+ = addition # you can also use it as +=
- = subtraction # you can also use it as -=
* = multiplication # you can also use it as *= same for all
/ = division # gives float ( 2/1 = 2.0)
% = modulus #gives the leftover after division
** = exponentiation #e.g.: 2 ** 4 means 2 * 2 * 2 * 2
// = floor division
```

- Variable math operator = number gives the same result as Variable = Variable math operator number.
- Variable += number gives the same result as Variable = Variable + number.

e.g.: a = a + 1 same as a += 1

- Any mathematics operator can be used before the = character to make an in-place operation:

```
-= decrements the variable in place,
+= increment the variable in place,
*= multiply the variable in place,
/= divide the variable in place,
//= floor divide the variable in place,
%= returns the modulus of the variable in place,
**= raise to power in place.
```

Input:

```
print(4 + 11) # sum of integers gives integer
```

Output:

```
15
```

Input:

```
print(39 + 1.0) # sum of an integer and float gives float
```

Output:

```
40.0
```

Input:

```
no1, no2 = 46, 52
no3 = no1 - no2
print(no3)
```

Output:

```
-6
```

Input:

```
no1 = 46
```

```
print(no1/23) # division gives float
```

Output:

```
2.0
```

Input:

```
print((3 * 4)/2) # parentheses are used as in normal mathematics operations
```

Output:

```
6.0
```

Input:

```
print(7 // 2) # it gives integer part of division
```

Output:

```
3
```

Input:

```
print(9 % 2) # remainder of this division is 1
```

```
# it means 9 is an odd number
```

Output:

```
1
```

Input:

```
print(3**2)
```

Output:

```
9
```

Input:

```
print(64**0.5) # square root
```

Output:

```
8.0
```

Input:

```
print('Result of this (12+7) sum :', 12 + 7)
```

Output:

```
Result of this (12+7) sum : 19
```

Input:

```
pi = 3.14
```

```
r = 5
```

```
area = pi * r**2
```

```
print(area)
```

Output:

```
78.5
```

Input:

```
-print(2 ** 3) #2 to the power of 3
```

```
-print(3 ** 2)
```

```
-a = 2
```

```
-b = 8
```

```
-print( (a * b) ** 0.5 #square root
```

Output:

```
8
```

```
9
```

```
4.0 (bir sayının 0.5 kuvveti karekökünü verir)
```

Operations with 'print()' Function:

- When using print() we can write more than one expression in parentheses separated by “,”

Input:

```
number = 2020
text = "children deserve respect as much as adults in"
print(text, number)
```

Output:

```
children deserve respect as much as adults in 2020
```

#expressions are joined to each other by spaces
#the default value of keyword argument sep in the print() function

Input: print("yesterday I ate", 2, "apples")

Output: yesterday I ate 2 apples

The print() command automatically switches to the next line. This is due to the keyword argument end = "\n"
Here are the keyword arguments that run in the background of the print() function :

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False) # “\n” represents next line.

Input:

```
print('i, 'will say' end=' ')      #using end = ' ' we can combine expressions with space
print("i missed you", end=' ')    # If we didn't use end = ' ', we would normally get 3 lines of output.
print('to my mother')
```

Output:

```
i will say 'i missed you' to my mother
```

Input: print('smoking', 'is', 'slowly', 'killing me', sep=' + ')

Output: smoking + is + slowly + killing me

Input:

```
x = 5
print ('value of x      : ', x)
```

x += 2

```
print ("2 more of x    : ", x, "\n") # using '\n', we produce extra line, so that we had empty line.
```

y = 10

```
print ('value of y    : ', y)
```

y -= 2

```
print ("2 minus y     : ", y, "\n")
```

z = 6

```
print ('value of z    : ', z)
```

z *= 2

```
print ("2 times z     : ", z, "\n")
```

Output:

```
value of x      : 5
2 more of x     : 7
```

```
value of y      : 10
```

```
2 minus y       : 8
```

```
value of z      : 6
```

```
2 times z       : 12
```

Input:

```
fruit = 'Orange'
vegetable = "Tomato"      #Remember. There is no difference between ' ', " " or "" "" "" ""
print (fruit, """" and """" , vegetable)
```

Output:

```
Orange and Tomato
```


Escape Sequences:

\ is a special sign used in expressions called **escape sequences**, which behaves according to the character immediately after \.

Here are basic escape sequences in Python:

- \n : means new line,
- \t : means tab mark, *#four spaces*
- \b : means backspace. *#It moves the cursor(imleç) one character to the left. (Deletes one thing before it)*

Input:

```
print('C:\\november\\number_expenditure.txt') #using \ before an escape sequence, it takes the magic of it
```

Output:

```
C:\november  
umber_expenditure.txt
```

Input:

```
print("one", "two", "three", sep="\t") # separated by tab marks
```

Output:

```
one    two    three
```

Input:

```
print('we', '\bare', '\bunited') # remember, normally print() function  
                                # separates expressions by spaces  
                                # \b is an escape sequence and it deletes one thing before it
```

Output:

```
weareunited
```

Normally when we use ' inside the ', Python will give error. Because single-quote in single quotes gives an error. But here, in the example below, \ allows single-quote ' to be ignored (\ takes the magic after it).

So it gives no error.

Be careful, when using \ in the long string. It may cause error because of its functionality.

Using \\ guarantees no error.

Input:

```
print('it\'s funny to learn Python')
```

Output:

```
it's funny to learn Python
```

Boolean Operations:

As we learned in the previous lesson boolean or bool can only have two values. **True** and **False**. We can say that bool represent 1 or 0, yes or no, exist or nonexistent.

Python has three built-in boolean operators: **and**, **or**, **not**. Except not, all are binary operators, which means two arguments are required.

And operator: It evaluates all expressions and returns the last expression if all expressions are evaluated **True**. Otherwise, it returns the first value that evaluated **False**.

#Takes the last one if all is True, otherwise takes the first False

Or operator: It evaluates the expressions left to right and returns the first value that evaluated True or the last value (if there is no True).

#Takes the first True. If there is no True, takes the last value

Not operator: For example : `x = True and not True`, the value of `x` returns **False**. `#not True means False`

- **Order of Priority:** 1- **not** 2- **and** 3- **or**

e.g.:

Input:

```
logic = True and False or not False or False  
print(logic)
```

Output:

True

- **and** and **or** return one of its operands, not necessarily a **bool** type. But **not** always returns **bool** type.

Truth Values of Logic Statements:

- The values of non-boolean types (integers, strings, etc.) are considered **truthy** or **falsy** when used with logical operations, depending on whether they are seen as **True** or **False**.
- The following values are considered **False**, in that they evaluate to False when applied to a boolean operator:

-None.

-Zero of any numeric type: 0, 0.0, 0j

-Empty sequences and collections: "", ' ', [], {}, ()

-Other than above values, any remaining value is evaluated as **True**.

Input: `print(2 and 3)` # takes the last one if all is true

Output: 3

Input: `print(1 and 0)` # takes the first False

Output: 0

Input: `print([] and "Hello World")`

Output: []

Input: `print(2 or 3)` #takes the first True

Output: 2

Input: `print(None or {})` #takes the last False

Output: {}

Input: `print([] or "Hello World")` #takes the first True

Output: Hello World

Input: `print(0 or 1 or 2 or "")`

Output: 1

Input: `not 1`

Output: False

The Strength of Strings in Python

Indexing&Slicing Strings

You can access all elements of a string type data very easily. Accordance with the sequence of string letters, you can specify them from left to right in brackets.

Input:

```
fruit = 'Orange'
```

```
print('Word          : ', fruit)
print('First letter   : ', fruit[0])
print('Second letter  : ', fruit[1])
print("3rd to 5th letters : ", fruit[2:5])
print("Letter all after 3rd : ", fruit[2:])
```

Output:

```
Word          : Orange
First letter   : O
Second letter  : r
3rd to 5th letters : ang
Letter all after 3rd : ange
```

- Remember, the enumeration of a string starts from **zero**.

The formula syntax of string indexing is : **string[start:stop:step]**.

string[:] : returns the full copy of the sequence

string[start:] : returns elements from start to the end element

string[:stop] : returns element from the 1st element to stop-1

string[::step] : returns each element with a given step

e.g. :

Input:

```
city = 'Phoenix'
```

```
print(city[1:]) # starts from index 1 to the end
print(city[:6]) # starts from zero to 5th index -it stops before one character (whet it sees it)
print(city[::2]) # starts from zero to end by 2 step
print(city[1::2]) # starts from index 1 to the end by 2 step
print(city[-3:]) # starts from index -3 to the end
print(city[::-1]) # negative step starts from the end to zero 2 step
print(city[1:-3]) # starts from index one to the negative three
```

Output:

```
hoenix          # city[1:5] and city[-6:-2] both gives hoen
Phoeni
Ponx
hei
nix
xineohP
hoe
```

- The **len()** function gives the length (number of characters)

Input:

```
vegetable = 'Tomato'
```

```
print('length of the word', vegetable, 'is :', len(vegetable))
```

Output:

```
length of the word Tomato is : 6
```

String Formatting with Arithmetic Syntax:

- Arithmetic syntax (+, *, and =),
- % operator formatting,
- string.format() method,
- f-string formatting.

A method is like a function, except it is attached to an object. We call a method on an object, and it possibly makes changes to that object (like `string.format()`). A method, then, belongs to a class.

Arithmetic syntax (+, =, *) :

- We can use + operator for combining the two string together without any spaces.

Input: `print('clarus' + 'way')`

Output: clarusway

- We can also use * operator for repeating the string without any spaces.

Input: `print(3*'no way !')`

Output: no way!no way!no way!

Input:

```
fruit = 'Orange'
vegetable = 'Tomato'
print("using + :", fruit + vegetable)
print("using * :", 3 * fruit)
```

Output:

```
using + : OrangeTomato
using * : OrangeOrangeOrange
```

- As with numeric types, we can do addition operation in-place either with string type using +=.

Input:

```
fruit = 'orange'
fruit += ' apple'  #there is one space
```

```
print(fruit)
```

Output:

```
orange apple
```

Input:

```
fruit = 'orange'
fruit += ' apple'  #there is a space at the begining
fruit += ' banana'. #it would give the output all as one if there wasn't spaces
fruit += ' apricot'
print(fruit)
```

Output:

```
orange apple banana apricot
```

String Formatting with '%' Operator: (derste işlemedik. Sadece preclassta)

- `%` operator gets the values in order and prints them in order using several characters accordingly.
For now, we used only `s`, `d` and `f` characters to specify the data type in a string.

Input:

```
phrase = 'I have %d %s and %.2f brothers' % (4, "children", 5)
print (phrase)
```

Output:

I have 4 children and 5.00 brothers

Here in the example, the `%` operator first takes '4' and puts it in the first `%` operator, then takes 'children' secondly and puts it in the second `%` operator and finally takes '5' and puts it in the third `%` operator.

- In the `'%s'` syntax : `s` stands for 'string'.
- In the `'%.2f'` syntax : `f` stands for 'float'. In this example 2 digits after point.
- In the `'%d'` syntax : `d` stands for 'numeric'.

If you want, you can limit the character numbers of the strings.

Input:

```
sentence = "apologizing is a virtue"
```

```
print("%.11s" % sentence) # we get first 11 characters of the string
```

Output:

apologizing

You can also use variables with `%` operator to format the string. Let's look at the example :

Input:

```
print('%(amount)d pounds of %(fruit)s left' % {'amount': 33, 'fruit': 'bananas'})
```

Output:

33 pounds of bananas left

In this example, we used two variables which are `amount` and `fruit`. If you noticed, we assign values to variables in curly braces `'{'`. This format is a dictionary type that you will learn in the next lessons.

String Formatting with 'string.format()' Method:

You can make strings change depending on the value of a variable or an expression.

The value of expression comes from `.format()` method in order. Curly braces `{}` receives values from `.format()`.

Input:

```
fruit = 'Orange'
vegetable = 'Tomato'
amount = 4
print('The amount of {} we bought is {} pounds'.format(fruit, amount))
```

Output:

The amount of Orange we bought is 4 pounds

- If you've written more variables than you need in the `.format()` method, the extra ones just will be ignored. Using keywords in `{}` makes string more readable. For example:

Input:

```
print('{state} is the most {adjective} state of the {country}'.format(state='California', country='USA', adjective='crowded'))
```

Output:

California is the most crowded state of the USA

- If you have noticed, we do not have to write the keywords in `.format()` method in order.

You can combine both positional and keyword arguments in the same `.format()` method :

At this point, let us give you some explanations : Positional arguments are arguments that can be called by their position in the function or method definition. Keyword arguments are arguments that can be called by their names.

Input:

```
print('{0} is the most {adjective} state of the {country}'.format('California', country='USA', adjective='crowded'))
```

Output:

California is the most crowded state of the USA

- You can use the same variable in a string more than once if you need it. Also, you can select the objects by referring to their positions in brackets.

Input:

```
print("{6} {0} {5} {3} {4} {1} {2}".format('have', 6, 'months', 'a job', 'in', 'found', 'I will'))
```

Output:

I will have found a job in 6 months

- Be careful not to write keyword arguments before positional arguments.

Using `str.format()` method is much more readable and useful than using `%-operator` formatting in our codes, but `str.format()` method can still be too wordy if you are dealing with multiple parameters and longer strings. At this point, the `f-string` formatting which you will learn in the next lesson suffices.

The order of the words :('in', 'know', 'bring', 'to', 'students.', 'out', 'best', 'teachers', 'the', 'Good', 'how')

Input:

```
print("{9} {7} {1} {10} {3} {2} {5} {8} {6} {0} {4}".format('in', 'know', 'bring', 'to', 'students.', 'out', 'best', 'teachers', 'the', 'Good', 'how'))
```

Output:

Good teachers know how to bring out the best in students.

String Formatting with 'f-string':

It is the easiest and useful formatting method of the strings.

It makes string formatting easier.

f-string is the string syntax that is enclosed in quotes with a letter f at the beginning. Curly braces {} that contain variable names or expressions are used to replace with their values.

Sample of a formula syntax is : **f"strings {variable1} {variable2} string {variable3}"**

Input:

```
fruit = 'Orange'
vegetable = 'Tomato'
amount = 6
output = f"The amount of {fruit} and {vegetable} we bought are totally {amount} pounds"
print(output)
```

Output: The amount of Orange and Tomato we bought are totally 6 pounds

- You can use all valid expressions, variables, and even methods in curly braces.

Input:

```
result = f"{4 * 5}"
print(result)
```

Output: 20

Input:

```
my_name = 'JOSEPH'
output = f"My name is {my_name.capitalize()}"
print(output)
```

Output: My name is Joseph

Input:

```
name = "Joseph"
job = "teachers"
domain = "Data Science"
message = (
    f"Hi {name}. "
    f"You are one of the {job} "
    f"in the {domain} section."
)
print(message)
```

Output: Hi Joseph. You are one of the teachers in the Data Science section.

- If you want to use multiple **f-string** formatting lines without parentheses, you will have the other option that you can use backslash \ between lines.

Input:

```
name = "Joseph"
job = "teachers"
domain = "Data Science"
message = f"Hi {name}. " \
    f"You are one of the {job} " \
    f"in the {domain} section."
print(message)
```

Output: Hi Joseph. You are one of the teachers in the Data Science section.

Main String Operations

Searching a String:

To search patterns in a string there are two useful methods called `startswith()` and `endswith()` that search for the particular pattern in the immediate beginning or end of a string and return **True** if the expression is found. Here are some simple examples. Examine the basic syntax of those methods carefully.

Input:

```
text = 'www.clarusway.com'
print(text.endswith('.com'))
print(text.startswith('http:'))
```

Output:

```
True
False
```

Input:

```
text = 'www.clarusway.com'
print(text.endswith('om'))
print(text.startswith('w'))
```

Output:

```
True
True
```

The formula syntaxes are :

- `string.startswith(prefix[, start[, end]])`
- `string.endswith(suffix[, start[, end]])`

Input:

```
email = "clarusway@clarusway.com is my e-mail address"
print(email.startswith("@", 9))
print(email.endswith("-", 10, 32))
```

Output:

```
True
True
```

Changing a String:

The methods described below return the copy of the string with some changes made.

How does the following syntax work?

A string is given first (or the name of a variable that represents a string), then comes a period followed by the method name and parentheses in which arguments are listed.

The formula syntax is : `string.method()`

Let's examine some common and the most important methods of string changing :

`str.replace(old, new[, count])` replaces all occurrences of old with the new.

The count argument is optional, and if the optional argument count is given, only the first count occurrences are replaced. **count**: Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

`str.swapcase()` converts upper case to lower case and vice versa.

`str.capitalize()` changes the first character of the string to the upper case and the rest to the lower case.

`str.upper()` converts all characters of the string to the upper case.

`str.lower()` converts all characters of the string to the lower case.

`str.title()` converts the first character of each word to upper case.

Let's consolidate the subject through examples :