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_syntx
-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”

Data Types and Useful Operations

—Data Types:

1-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

2-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.

3-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
```

—Operations:

1-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 devision

• 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) # 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 like mathematics
```
Output:
6.0

Input:
```
print(7 // 2)  # it gives integer part of division
```
Output:
3

Input:
```
print(9 % 2)
```
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)

2-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
```

3-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: not, and, or. Except not, all are binary operators, which means two arguments are required.

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

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

- **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 (when 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
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 ! ') # same: print('no way ! '*3)
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.
Print
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.
e.g:

Input:

```
sentence = "I live and work in Virginia"
print(sentence.upper())
print(sentence.lower())
print(sentence.swapcase())
print(sentence)  # note that, source text is unchanged
```

Output:

```
I LIVE AND WORK IN VIRGINIA
i live and work in virginia
i LIVE AND WORK IN vIRGINIA
I live and work in Virginia
```

If we assign the modified text to a new variable, we can have a new string. Consider these:
Input:
```
sentence = "I live and work in Virginia"
title_sentence = sentence.title()
print(title_sentence)

changed_sentence = sentence.replace("i", "+")
print(changed_sentence)

print(sentence)  # note that, again source text is unchanged
```
Output:
```
I Live And Work In Virginia
I l+ve and work +n V+rg+n+a
I live and work in Virginia
```

Input:
```
sentence = "I live and work in Virginia"
swap_case = sentence.swapcase()
print(swap_case)
print(swap_case.capitalize())  # changes 'i' to uppercase and the rest to lowercase
```
Output:
```
i LIVE AND WORK IN vIRGINIA
I live and work in virginia
```

Editing a String:
The methods described below remove the trailing characters (i.e. characters from the right side). The default for the argument chars is also whitespace. If the argument chars aren't specified, trailing whitespaces are removed.
The formula syntax is : **string.method()**
- **str.strip()** : removes all spaces (or specified characters) from both sides.
- **str.rstrip()** : removes spaces (or specified characters) from the right side.
- **str.lstrip()** : removes spaces (or specified characters) from the left side.

Input:
```
space_string = "     listen first      "
print(space_string.strip())  # removes all spaces from both sides
```
Output:
```
listen first
```
Input:
```
source_string = "interoperability"
print(source_string.strip("yi"))     # removes trailing "y" or "i" or "yi" or "iy" from both sides
```
Output:
```
nteroperabilit
```
Input:
```
source_string = "interoperability"
print(source_lstring.strip("in"))     # removes "i" or "n" or "in" or "ni" from the left side
```
Output:
```
teroperability
```
Input:
```
space_string = "     listen first      "
print(space_string.rstrip())  # removes spaces from the right side
```
Output:
```
     listen first
```

Input:
```
source_string = "interoperability"
print(source_string.rstrip("yt"))  # removes "y" or "t" or "yt" or "ty" from the right side
```
Output:
```
source_string = "interoperability"
print(source_string.rstrip("yt"))
```

Collection Types

There are various collection types in Python. While types such as int and str hold a single value, collection types hold multiple values.
In your programs, you usually need to group several items to render as a single object. We use collection types of data to do this job.

1-List:

One of the most useful collections in Python is a list. In Python, a list is only an ordered collection of valid Python values.
The list type is probably the most commonly used collection type in Python. In spite of its name, a list is more like an array in some other languages (e.g. JavaScript).

Creating a List:
A list can be created by enclosing values, separated by commas, in square brackets [].
Let's create a simple list that includes some country names.

Input:
```
country = ['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New Zealand']

print(country)
```
Output:
```
['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New Zealand']
```

- All the country names are printed in the same order as they were stored in the list because lists are ordered.

Another way to create a list is to call the 'list()' function.
You do this when you want to create a list from an iterable object: that is, type of object whose elements you can import individually. The lists are iterable like other collections and string types.

Input:
```
string_1 = 'I quit smoking'

new_list_1 = list(string_1)  # we created multi element list
print(new_list_1)

new_list_2 = [string_1]  # this is a single element list
print(new_list_2)
```
Output:
```
['I', ' ', 'q', 'u', 'i', 't', ' ', 's', 'm', 'o', 'k', 'i', 'n', 'g']
['I quit smoking']
```

- Note that, using **list()** function, all characters of string_1 including spaces was moved into a new_list_1.
- If you noticed, lists can contain more than one of the same value.

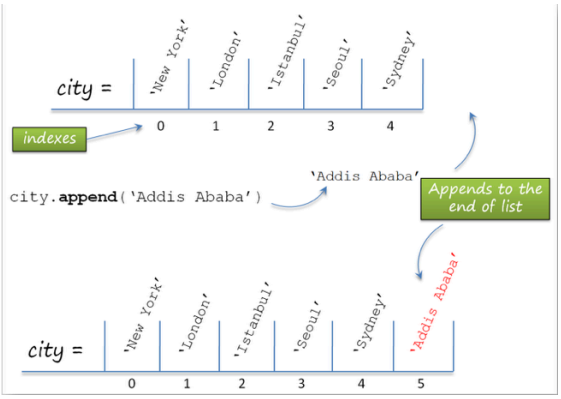
As it appears, the list() function creates a list that contains each component of a specific iterable object, such as a string. You can use square brackets or list() functions, depending on what you are going to do.
The components of a list are not limited to a single data type, given that Python is a dynamic language: e.g.
mixed_list = [11, 'Joseph', False, 3.14, None, [1, 2, 3]]

Basic Operations with Lists:

There are many methods and functions for dealing with the list structures. In most cases, we'll have to make an empty list to fill it later with the data you want.
-empty_list_1= []
-empty_list_2 = list()
We can add an element into a list using .append() or .insert() methods.

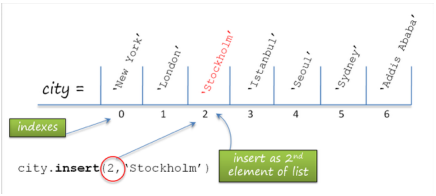
.append() : Append an object to end of a list. Using only list.append(element) syntax, returns none. If you want to see the new appended list, you have to call or print it. See the example :

Input:
```
empty_list_1 = []
empty_list_1.append('114')
empty_list_1.append('plastic-free sea')
print(empty_list_1)
```
Output:
['114', 'plastic-free sea']
Input:
```
city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']
city.append('Addis Ababa')
print(city)
```
Output:
['New York', 'London', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']



.insert() : Add a new object to list at a specific index. The syntax looks like list.insert(index, object).

Input:
```
city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']
city.insert(2, 'Stockholm')
print(city)
```
Output:
['New York', 'London', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']



- We can remove the elements in lists using **list.remove()** method or sort the elements using **list.sort()**

Input:
```
city = ['New York', 'London', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']
city.remove('London')
print(city)  # we have deleted 'London'
```
Output:
['New York', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']
Input:
```
city = ['New York', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']
city.sort()  # lists the items in alphabetical order
print(city)
```
Output:
['Addis Ababa', 'Istanbul', 'New York', 'Seoul', 'Stockholm', 'Sydney']

- Likewise, the length of the list elements can be calculated with the len() function also. Let's calculate the length of 'city' variable we have.

Input:
```
city = ['Addis Ababa', 'Istanbul', 'New York', 'Seoul', 'Stockholm', 'Sydney']
print(len(city))
```
Output:
6

- One of the important operations of the lists is assigning an element to the specific index number.

Input:
```
city = ['New York', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']
city[1] = 'Melbourne'  # we assign 'Melbourne' to index 1
print(city)
```
Output:
['New York', 'Melbourne', 'Istanbul', 'Seoul', 'Sydney', 'Addis Ababa']

Accessing Lists:

There are several types of collections for storing data in Python: list, tuple, dictionary.
Each item or element in a list, as well as every character in a string, has an index corresponding to their location. Using indexes, we can access elements within a sequence.

Indexing a List:

If we want to access or use the elements of a list, we can do that using index numbers of the list enclosed by square brackets.

Input:
```
colors = ['red', 'purple', 'blue', 'yellow', 'green']
print(colors[2])  # If we start at zero, the second element will be ‘blue'.
```
Output:
```
blue
```

Input:
```
city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']
city_list = []
city_list.append(city)   # we have created a nested list. (A list in a list)
print(city_list)
```
Output:
```
[['New York', 'London', 'Istanbul', 'Seoul', ‘Sydney’]]
```

city_list includes only one element which is the city list.

Input:
```
city_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]
print(city_list[0])   # access to first and only element
```
Output:
```
['New York', 'London', 'Istanbul', 'Seoul', ‘Sydney']
```

'city_list[0]' is a list type data. So that, we can still access its elements via indexing. Let's access its second element :

Input:
```
city_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]
print(city_list[0][2])  # second element of the zeroth element
```
Output:
```
Istanbul
```

- 'city_list[0][2]' is a string type data. So, we can also access its elements via indexing.

Input:
```
city_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]
print(city_list[0][2][3])  #third element of the second element of the zeroth element
```
Output:
```
a
```

Slicing a List:
We can access individual elements of a list, as well as part of those items. We use index numbers again for slicing but we do it by typing it a little differently.

Input:
```
numbers = [1, 3, 5, 7, 9, 11, 13, 15, 17]
print(numbers[2:5])  # we get the elements from index=2 to index=5(5 is not included)
```
Output:
```
[5, 7, 9]
```

The formula syntax is : sequence[start:stop:step]
This formula produces a slice of the sequence where start is an index of the first element required (the element is included in the slice) and stop is an index of the end element (the element is not included in the slice), step is an interval between elements to be chosen.
In this example, we will create a list of numbers from 1 to 10 using 'range()' function and select even ones:
Input:
```
count = list(range(11))
print(count)
print(count[0:11:2])
```
Output:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 2, 4, 6, 8, 10]
```

By the way, range() function returns an object that produces a sequence of integers from start (including) to stop (excluding) by step.
The formula syntax is : range(start, stop[, step])
Each part of the slice has a default value, so they are optional. If we don't assign a value to the start index, it is considered to be 0; if we don't assign a value to the stop index, it will be the same as the length of the sequence.
- my_list[:]: returns the full copy of the sequence
- my_list[start:] : returns elements from start to the end element
- my_list[:stop] : returns element from the 1st element to stop-1
- my_list[::step] : returns each element with a given step

Let's do some more examples to grasp it.

The following example outputs the same as the input list (animals).
Input:
```
animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']
print(animals[:])  # all elements of the list
```
Output:
```
['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', ‘giraffe’]
```

Input:
```
animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']
print(animals[3:])
```
Output:
```
['wolf', 'rabbit', 'deer', ‘giraffe']
```

The following example slices the animals starts at index=0 to the index=4.
Input:
```
animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']
print(animals[:5])
```
Output:
```
['elephant', 'bear', 'fox', 'wolf', ‘rabbit']
```
And the last example slices animals starts at index=0 to the end with 2 step.
Input:
```
animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']
print(animals[::2])
```
Output:
```
['elephant', 'fox', 'rabbit', 'giraffe']
```

Negative Indexing & Slicing:
Negative indexing is the best and shortest way to reach the elements at the end of the list. The negative indexing works in reverse. We can reach the last element of a list as list_name[-1]. See the example below :
Input:
```
city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']
print(city[-4])
```
Output:
London

Negative slicing also works similarly, as we see in single element access. In this case, step index can also be negative. If the step index is negative the elements of sequence will return in reverse order. Let's see in examples :

Input:
```
reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']
print(reef[-3:])
```
Output:
['lobster', 'squid', ‘octopus']

Input:
```
reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']
print(reef[:-3])
```
Output:
['swordfish', 'shark', 'whale', ‘jellyfish']

Input:
```
reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']
print(reef[::-1])  # we have produced the reverse of the list
```
Output:
['octopus', 'squid', 'lobster', 'jellyfish', 'whale', 'shark', ‘swordfish']

Input:
```
reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']
print(reef[::-2])
```
Output:
['octopus', 'lobster', 'whale', ‘swordfish']

Input:
```
odd_no = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(odd_no[7:3:-1])
print(odd_no[2:6:-1])
```
Output:
[8, 7, 6, 5]
[]

2-Tuples:

A tuple is another collection type that can hold multiple data very similar to the list.
The most important difference from the list is that the tuple is immutable. Therefore, methods like append() or remove() do not exist in the operations of this type.
Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port. If we have unchanged data, we should choose tuples because it is much faster than lists.
We used square brackets '[]' to define the lists. In the tuple, normal parentheses '()' are used.
The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid.

Difference between list and tuple?
LISTs :
- Lists are mutable i.e they can be edited.
- Lists are slower than tuples.
- Syntax: list_1 = [True, 'Space', 20]

TUPLEs :
- Tuples are immutable # tuples are lists which can't be edited
- Tuples are faster than list.
- Syntax: tup_1 = (True, 'Space' , 20)

Creating a Tuple:

A tuple also can be created by enclosing values, separated by commas, in parentheses.
You can compare tuple to a case. When you put the data that you want it to not change and close the lid, you can no longer change this data, modify its size and edit it.

Input:
```
empty_tuple = ()
print(type(empty_tuple))
```
Output:
```
<class 'tuple'>
```

If you want to create a single element tuple, you should use a comma.

Input:
```
try_tuple = ('love')
print(try_tuple)
print(type(try_tuple)) # it's not tuple type.
```
Output:
```
love
<class 'str'>
```

It occurs in only single element tuples and we can fix the problem using comma at the end of the element.
Tips:
- Remember to always use a comma when defining a singleton tuple.

Input:
```
try_tuple = ('love',)
print(try_tuple)
print(type(try_tuple)) # it's a tuple type.
```
Output:
```
('love',)
<class 'tuple'>
```

Actually, if your tuple contains more than one element, separating elements with commas will be enough. Another way to create a tuple is to call the tuple() function. You do this when you want to create a tuple from an iterable object: that is, a type of object whose elements you can import individually.
The tuple is also iterable like other collections and string types. Let's create another tuple using tuple() function. With this function, you can create an empty tuple as well.

Input:
```
planets = 'mercury', 'jupiter', ‘saturn'

print(planets)
print(type(planets))
```
Output:
```
('mercury', 'jupiter', 'saturn')
<class ‘tuple’>
```

Input:
```
empty_tuple_1 = tuple()

print(empty_tuple_1)
print(type(empty_tuple_1))
```
Output:
```
()
<class ‘tuple'>
```

It is easy to convert between list and tuple as in the examples below :

Input:
```
my_tuple=(1, 4, 3, 4, 5, 6, 7, 4)

my_list = list(my_tuple)

print(type(my_list), my_list)
```

Output:
```
<class 'list'> [1, 4, 3, 4, 5, 6, 7, 4]
```

Input:
```
my_list = [1, 4, 3, 4, 5, 6, 7, 4]

my_tuple = tuple(my_list)

print(type(my_tuple), my_tuple)
```

Output:
```
<class 'tuple'> (1, 4, 3, 4, 5, 6, 7, 4)
```

An iterable string can be converted to a tuple :
Input:
```
mountain = tuple('Alps')
print(mountain)
```
Output:
```
('A', 'l', 'p', 's')
```

How can We Use a Tuple ?
If you want, let's take a look at the common features of the list and tuple. So you can have an idea of what to do with tuples.
Both lists and tuples are ordered. It means that when storing elements to these containers, you can sure that their order will remain the same. You can also duplicate values or mix different data types in tuples.

Input:
```
mix_value_tuple = (0, 'bird', 3.14, True)

print(len(mix_value_tuple))
```

Output:
4

As we stated at the beginning, just like lists, tuples support indexing :

Input:
```
even_no = (0, 2, 4)
print(even_no[0])
print(even_no[1])
print(even_no[2])
print(even_no[3])
```
Output:
0
2
4
---
print(even_no[3]) : IndexError: tuple index out of range

And one of the most important differences of tuples from lists is that 'tuple' object does not support item assignment. Yes, because tuple is immutable. See the example :

Input:
```
city_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']

city_list[0] = 'Athens'
city_list[1] = 'Cairo'
print(city_list)
```
Output:
['Athens', 'Cairo', 'Moskow', ‘Dublin']

Input:
```
city_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']

city_tuple = tuple(city_list)

city_tuple[0] = 'New York'  # you can't assign a value
```
Output:

TypeError: 'tuple' object does not support item assignment

Benefits of Immutability
- Tuples are faster and more powerful in-memory than lists. You should give it a thought whenever you need to deal with large amounts of data. If you don't want to change your data you may have to choose tuples.
- Because of its immutability, the data stored in a tuple can not be altered by mistake.
- A tuple can be used as a dictionary (we will see in the next lesson) key, while 'TypeError' can result in lists as keys. And this is the usefulness of tuples in the data processing.

3-Dictionaries

In this topic, we will examine the collection types which store item pairs. What does it mean?
Think of a real dictionary. It contains words and their meanings. In Python, you can accept the words as key and the meaning of the words as value.
A dictionary in Python is a collection of key-value pairs called items of a dictionary. The dictionary is enclosed by curly braces {}. Each pair (item) is separated by a comma and the key and value are separated by a colon.

Creating a Dictionary

A dictionary also can be created by enclosing pairs, separated by commas, in curly-braces. Looks like list or tuple, right?
And of course, we can use a function to create a dictionary : 'dict()' function. Let's create a simple empty dict :

```
empty_dict_1 = {}

empty_dict_2 = dict()
```

This is our first dict in this lesson. Now let's print its type.

Input:
```
empty_dict_1 = {}

print(type(empty_dict_1))
```
Output:
```
<class 'dict'>
```

The basic form of dict looks like :

```
my_dict = {'key1': 'value1',
           'key2': 'value2',
           'key3': 'value3'
           }
```

The syntax for accessing an item is very simple. We write a key that we want to access in square brackets. This method works both for adding items to a dict and for reading them from there.
In the following examples, you'll see several methods that allow us to create a dict and add a key-value pair to it.

Input:
```
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

print(state_capitals['Colorado']) # accessing method
```
Output:
```
Denver
```

Input:
```
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

state_capitals['Virginia'] = 'Richmond' # adding a new item

print(state_capitals)
```
Output:
```
{'Arkansas': 'Little Rock',
'Colorado': 'Denver',
'California': 'Sacramento',
'Georgia': 'Atlanta',
'Virginia': 'Richmond'}
```

- Note that keys and values can be of different types.

```
mix_values = {'animal': ('dog', 'cat'),  # tuple type
              'planet': ['Neptun', 'Saturn', 'Jupiter'],  # list
              'number': 40,  # int type
              'pi': 3.14,  # float type
              'is_good': True}  # bool type
```

```
mix_keys = {22 : "integer",
            1.2 : "float",
            True : "boolean",
            "key" : "string"}
```

And now, let's use dict() function to create a dictionary :

Input:

```
dict_by_dict = dict(animal='dog', planet='neptun', number=40, pi=3.14, is_good=True)
print(dict_by_dict)
```

Output:

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True}
```

- Do not use quotes for keys when using the dict() function to create a dictionary.
- You cannot use iterables as keys to create a dictionary.

Main Operations with Dictionaries

There are several methods that allow us to access items, keys, and values. You can access all items using the **.items()** method, all keys using the **.keys()** method, and all values using the **.values()** method:

Input:

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

print(dict_by_dict.items(), '\n')
print(dict_by_dict.keys(), '\n')
print(dict_by_dict.values())
```

Output:

```
dict_items([('animal', 'dog'), ('planet', 'neptun'),
        ('number', 40), ('pi', 3.14), ('is_good',
True)])

dict_keys(['animal', 'planet', 'number', 'pi',
'is_good'])

dict_values(['dog', 'neptun', 40, 3.14, True])
```

You have learned that you can add a new item by assigning value to a key that is not in the dictionary. Likewise, you can add new items using the **.update()** method. Let's see :

Input:

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

dict_by_dict.update({'is_bad': False})

print(dict_by_dict)
```

Output:

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True,
'is_bad': False}
```

You can also remove an item using the **del** function:
The formula syntax is : del dictionary_name[‘key’].

Input:

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}

del dict_by_dict['animal']

print(dict_by_dict)
```

Output:

```
{'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True,
'is_bad': False}
```

Using the **in** and the **not in** operator, you can check if the key is in the dictionary.

- When we use the **in** operator; if the key is in the dictionary, the result will be True otherwise False.
- When we use the **not in**; if the key is not in the dictionary, the result will be True otherwise False.

Input:

```
dict_by_dict = {'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}

print('pi' in dict_by_dict)
print('animal' not in dict_by_dict)  # remember, we have deleted ‘animal'
```

Output:

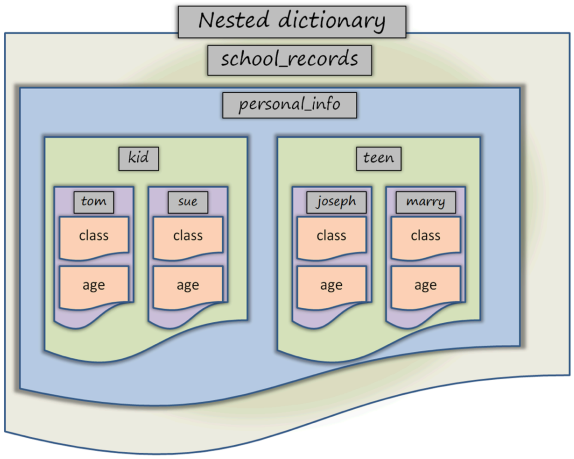
```
True
True
```

Nested Dictionaries

In some cases, you need to work with the nested dict. When you decide to specialize in data science, we will work very often with dictionaries in the future.

```
school_records={
   "personal_info":
      {"kid":{"tom": {"class": "intermediate", "age": 10},
            "sue": {"class": "elementary", "age": 8}
           },
       "teen":{"joseph":{"class": "college", "age": 19},
            "marry":{"class": "high school", "age": 16}
           },
      },

   "grades_info":
      {"kid":{"tom": {"math": 88, "speech": 69},
            "sue": {"math": 90, "speech": 81}
           },
       "teen":{"joseph":{"coding": 80, "math": 89},
            "marry":{"coding": 70, "math": 96}
           },
      },
}
```



We can use square brackets to access internal dicts :

Input:

```
school_records={
   "personal_info":
      {"kid":{"tom": {"class":"intermediate", "age":10},
            "sue": {"class":"elementary", "age":8}
           },
       "teen":{"joseph":{"class":"college", "age":19},
            "marry":{"class":"high school", "age":16}
           },
      },
}

print(school_records['personal_info']['teen']['marry']['age'])
```

Output:

```
16
```

4-Sets

A set is a collection of elements with no repeats and without insertion order but sorted order.
Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.
They can hold multiple data in them, but only one of value. They are used in situations where it is only important that some things are grouped together, and not what order they were included.

Creating a Set
Curly braces '{}' or the **set()** function can be used to create sets. But the only way to create an empty set is: use the set() function.

- Note that, to create an empty set you have to use **set()** function. Do not use {} to create an empty set. Otherwise, you will create an empty dictionary.

```
empty_set = set()
```

Input:

```
empty_set = set()

print(type(empty_set))
```

Output:

```
<class ‘set'>
```

We will now see how sets have unordered and unique objects.

Input:

```
colorset = {'purple', 'orange', 'red', 'darkblue', 'yellow', 'red'}

print(colorset)
print(colorset)
```

Output:

```
{'darkblue', 'orange', 'purple', 'red', 'yellow'}
{'darkblue', 'purple', 'orange', 'yellow', ‘red’}
```

As you can see in the output, the two 'red' values we have defined in the set have fallen to one. And every time you print the set, the order of the objects in the set changes.
Let's look at another example :

Input:

```
s = set('unselfishness')

print(s)
```

Output:

```
{'f', 'l', 'i', 'u', 'e', 'n', 'h', ’s'}
```

As you can see, the letters of the string type data are only written once in the set. Within this scope, using sets can help you avoid repetitions. Let's convert a list into a set and look at the repetitions of its elements:

Input:

```
flower_list = ['rose', 'violet', 'carnation', 'rose', 'orchid', 'rose', 'orchid']
flowerset = set(flower_list)
flowerlist = list(flowerset)

print(flowerset)
print(flowerlist)
```

Output:

```
{'orchid', 'carnation', 'violet', 'rose'}
['orchid', 'carnation', 'violet', 'rose']
```

Main Operations with Sets
There are several methods that allow us to add and remove items to/from sets. Moreover, we have the methods of intersection, unification, and differentiation of sets :
These methods are :

- **.add()** : Adds a new item to the set.
- **.remove()** : Allows us to delete an item.
- **.intersection()** : Returns the intersection of two sets.
- **.union()** : Returns the unification of two sets.
- **.difference()** : Gets the difference of two sets.

Input:
```
a = set('abracadabra')

print(a)
```
Output:
```
{'a', 'b', 'c', 'd', 'r'}
```

Input:
```
a = set('abracadabra')
b = set('alacazam')

print(a - b)  # same as '.difference()' method
print(a.difference(b)) # a difference from b
```
Output:
```
{'b', 'd', 'r'}
{'b', 'd', 'r'}
```

Input:
```
a = set('abracadabra')
b = set('alacazam')

print(a | b)  # same as '.union()' method
print(a.union(b)) # unification of a with b
```
Output:
```
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}
```

Input:
```
a = set('abracadabra')
b = set('alacazam')

print(a & b)  # same as '.intersection()' method
print(a.intersection(b)) # intersection of a and b
```
Output:
```
{'a', 'c'}
{'a', 'c'}
```

Input:
```
a = set('abracadabra')

a.remove('c') # we delete 'c' from the set
print(a)
```
Output:
```
{'a', 'b', 'd', 'r'}
```

Input:
```
a = set('abracadabra')

a.add('c') # we add 'c' again into the set
print(a)
```
Output:
```
{'a', 'b', 'c', 'd', 'r'}
```

Additionally, you can:

- Get the number of set's elements using **len()** function,
- Check if an element belongs to a specific set(in / not in operators), you get the boolean value.