PYTHON –DERS (27.09.2021 den itibaren)

Introduction to Data Types

Each data has a type, whether constant or variable. This type of data defines how you store it in memory and it also describes which process can be applied to it.

In fact, data types are nothing but variables you use to reserve some space in memory. Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

**💡Tips:**

* Note that, we assign value to a variable using 👉 **=**

You can think of **types** in the real world, bees are bee type and palm trees are palm type. That is, they have certain formats and common features.

We will now discuss some simple data types commonly used in Python:

* String,
* Signed Integer,
* Floating Point,
* Complex,
* Boolean.

**Strings**

Strings are identiﬁed as a contiguous set of characters represented in the quotation marks. Python allows for either pair of single or double (or even triple) quotes.

Strings are immutable sequence data type, i.e each time one makes any changes to a string, a completely new string object is created. You will be able to better understand **immutability** with examples that will continue in the next lessons.

If you want to work with any textual characters in your code, you have to work with strings. The string type is called str. Strings are the most common and useful data type in Python.

Carefully examine the following examples that will help you understand the type of string.

input :

text1 = "I have learned strings" # surrounded with double quotes

print(text1)

output :

I have learned strings

input :

e\_mail = 'joseph@clarusway.com' # surrounded with single quotes

print(e\_mail)

output :

joseph@clarusway.com

input :

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

output :

632

**Numeric Types**

For any programmer, using numbers is the most important issue. You can hardly write a serious program without using numbers, so let's talk about some basic numeric types. There are three distinct numeric types: **signed integer numbers**, **floating point numbers** and **complex numbers**.

* **Signed Integer** type is called int, they are whole numbers (positive, negative or zero), including no decimal point. For example: 71, -122, 0
* **Floating point** type is called float and they stand for real numbers with a decimal point. For example: 71.0, -33.03
* **Complex** type is called complex and they are written in the form, **x + yj** , where x is the real part and y is the imaginary part. For example: 3.14j. Imaginary numbers, also called complex numbers, are used in real-life applications, such as electricity, as well as quadratic equations. In quadratic planes, imaginary numbers show up in equations that don't touch the x-axis. Imaginary numbers become particularly useful in advanced calculus. We will not use this type much.

**💡Tips:**

* 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.

Q: What are the numerical data types in Python and their properties?  
A:

* **Integers :** they are whole numbers (positive, negative or zero), including no decimal point.
* **Floats :** they stand for real numbers with a decimal point.
* **Complexes :** they are written in the form, **x + yj** , where x is the real part and y is the imaginary part.

**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.

Bools are important data types that are widely used in Python as they can find use in every aspect of our daily lives. For example, imagine, whether the TV is turned on or off in your home or if the weather is rainy can be explained easily with bools.

Bools are mostly used in **conditional operations** which we will discuss in the next lessons.

tv\_open = True # it seems TV is on now

is\_rainy = False # I love sunny weather

Q: Describe the Boolean types in detail.  
A: **Boolean** types are called bool and their values are the two constant objects **True** and **False**. 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.  
  
**Bools** are important data types that are widely used in Python as they can find use in every aspect of our daily lives. For example, imagine, whether the TV is turned on or off in your home or if the weather is rainy can be explained easily with bools.

**Type Conversion**

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

* str() converts to **string** type
* int() converts to **signed integer** type
* float() converts to **floating point** type

**💡Tips:**

* We can print the types of data using type() function.

Look at the examples below to how we learn the types of data.

input :

example1 = 'sometimes what you say is less important than how you say it'

print(type(example1))

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'>

**💡Tips:**

* 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.

Here are some examples on converting between different types:

input :

f = 3.14 # the type is float

print(type(f))

output :

<class 'float'>

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')

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

**💡Tips:**

* Note that, it is important that the value of any type in Python can be converted to a **string**.

**Q**: What are the 'type conversion' and basic methods of that in Python?  
**A**: Type conversion refers to the conversion of one data type into another.  
  
**int()** – converts some data types into integer type.  
  
**float()** – converts some data types into float type.  
  
**str()** – converts any data type into string type.

example1 = 'sometimes what you say is less important than how you say it'

print(type(example1))

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'>

**💡Tips:**

* 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.

Here are some examples on converting between different types:

input :

f = 3.14 # the type is float

print(type(f))

output :

<class 'float'>

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')

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

**💡Tips:**

* Note that, it is important that the value of any type in Python can be converted to a **string**.

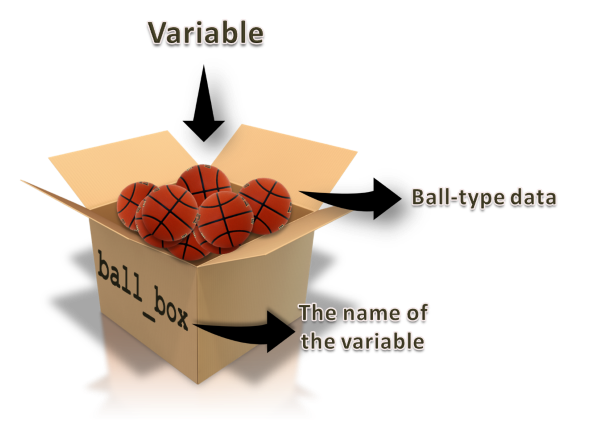
**Q**: What are the 'type conversion' and basic methods of that in Python?  
**A**: Type conversion refers to the conversion of one data type into another.  
  
**int()** – converts some data types into integer type.  
  
**float()** – converts some data types into float type.  
  
**str()** – converts any data type into string type.

**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



Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

To create a variable in Python, all you need to do is specify the variable name and then assign a value to it using 👉🏻**=**

The formula syntax of creating a variable and assigning a value to it is :

**variable name = value**

Remember, according to the PEP8, we had to give the variables a meaningful name for the data they kept inside.

Let's define variables and assign values to them :

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

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Solve this example with [**scratch**](https://scratch.mit.edu/projects/341546766/editor/).

**💡Tips:**

* Note that, the last value assigned to a variable is valid.

In Python, it is possible to assign the value of one variable to another variable:

input :

a = 5

b = 55

c = 555

c = a

b = c

a = b

print(a, b, c, sep=', ')

output :

5, 5, 5

**⚠️Avoid ! :**

* Note that, If you use undefined name of a variable in the code you write, you will get an 'NameError' message.

Q: What is the 'variable' and how do you assign a value to it?  
A: **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.  
  
Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.  
  
To create a variable in Python, all you need to do is specify the variable name and then assign a value to it.

**Simple Operations**

✔[Arithmetic Operations](https://lms.clarusway.com/mod/lesson/view.php?id=8&pageid=21)

✔[Operations with print( ) Function](https://lms.clarusway.com/mod/lesson/view.php?id=8&pageid=22)

✔[Escape Sequences](https://lms.clarusway.com/mod/lesson/view.php?id=8&pageid=23)

**Arithmetic Operations**

In Python, there are almost all of the arithmetic operations we use in mathematics. They are so simple to use and we can also use these operations on almost all data types, including string.

Basic Arithmetic operators are as follows :

| **Tiobe_Index** |
| --- |
| *Arithmetic Operators in Python* |

Let's now grasp the arithmetic operations with several examples:

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

**💡Tips:**

* We can assign a value to multiple variables. Consider this: variable1 = variable2 = 'clarusway opens your path'.
* We can also assign multiple values to multiple variables in sequence using commas as in example above.

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(2\*\*3)

output :

8

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

There is a list of priorities for all considered operations: it is worth keeping this priority in your mind.

1. parentheses : ()
2. power : \*\*
3. unary minus : -3
4. multiplication and division : \*, /
5. addition and subtraction : +, -

**Operations with 'print( )' Function**

Let's open a title here and take a closer look at print(), which is the most frequently used function. Since the need to make constant changes and see the results frequently occurs when writing code, printing directly on the screen can be the choice of most developers.

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

When using print() we can write more than one expression in parentheses separated by 👉🏻,

input :

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

output :

yesterday I ate 2 apples

When you type more than one expression in print(), you notice that the expressions are joined to each other by spaces. This is due to the default value of keyword argument **sep** in the print() function. This argument, which is defined as a **space** 👉🏻" " by default, is not visible in the background in the print().

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.
* Let's focus on the arguments sep= and end= we discussed above:

input :

print('i', end=' ')

print('will say', end=' ')

print("'i missed you'", end=' ')

print('to my mother')

output :

i will say 'i missed you' to my mother

If you noticed, in the example above we have combined all expressions with space in a single line using the 👉🏻end= If we didn't use end=' ', we would normally get 4 lines of output.

input :

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

output :

smoking + is + slowly + killing me

Some other useful operations that covers arithmetic and print() function are as follows. Carefully examine the examples:

input :

x = 5

print ('value of x : ', x)

x += 2

print ("2 more of x : ", x, "\n") # using string expression '\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

**💡Tips:**

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

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 :

fruit = 'Orange'

vegetable = "Tomato"

print (fruit, """ and """ , vegetable)

output :

Orange and Tomato

**💡Tips:**

* Remember. There is no difference between 👉🏻' ', 👉🏻" " or 👉🏻""" """.

**Escape Sequences**

Actually, the examples in the previous lesson show us how backslash 👉🏻\ works. 👉🏻\ 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,
* **\b** : means backspace. It moves the cursor one character to the left.

Look at these examples carefully:

input :

print('C:\\november\number\_expenditure.txt')

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

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. So it gives no error.

input :

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

output :

it's funny to learn Python

**⚠️Avoid ! :**

* Be careful, when using 👉🏻**\** in the long string. It may cause error because of its functionality described above. Using 👉🏻**\\** guarantees no error.

**Boolean Operations**

[Definitions](https://lms.clarusway.com/mod/lesson/view.php?id=10&pageid=24)

✔[Boolean Logic Expressions](https://lms.clarusway.com/mod/lesson/view.php?id=10&pageid=25)

✔[Order of Priority](https://lms.clarusway.com/mod/lesson/view.php?id=10&pageid=26)

✔[Truth Values of Logic Statements](https://lms.clarusway.com/mod/lesson/view.php?id=10&pageid=27)

Definitions

As we learned in the previous lesson boolean or bool can only have two values. True and False.

To put it easily, we can say that bool represent 1 and 0. In other words, yes & no or exist & nonexistent can be expressed by bool type.

For example, let's define a variable as to whether students have passed a course. Let the variable be called is\_pass. Then;

If you pass the course : is\_pass = True,  
If you did not pass the course : is\_pass = False

Q: What is a boolean in Python?  
A: Boolean is one of the built-in data types in Python, it mainly contains two values, and they are **True** and **False**.

**Boolean Logic Expressions**

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

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

d

**Or** operator : The or operator evaluates the expressions left to right and returns the ﬁrst value that evaluated True or the last value (if none is True).

| **Value1** | **Logic** | **Value2** | **Returns** |
| --- | --- | --- | --- |
| True | and | True | True |
| True | and | False | False |
| False | and | False | False |
| False | and | True | False |
| True | or | True | True |
| True | or | False | True |
| False | or | False | False |
| False | or | True | True |

Q: Python has three built-in Boolean operators. What are they?  
A: They are and, or, not.

**Order of Priority**

It is important to remember that, logical operators have a different priority and it has an effect on the order of evaluation. Here are the operators in order of their priorities:

1. **not**
2. **and**
3. **or**

For example : x = True and not True, the value of x returns False.

It evaluates not True first and gives False. It becomes x = True and False and gives False.

Let's consider one more example :

input :

logic = True and False or not False or False

print(logic)

output :

True

You can follow the logic priority flow in the diagram.

| **Tiobe_Index** |
| --- |
| *Bool Logic Priority* |

**💡Tips:**

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

Q: What is the order of priority of the logical operators?  
A:

1. not
2. and
3. or

**Truth Values of Logic Statements**

Although Python has its own boolean data type, we often use non-boolean values in logical operations.

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.

Here are some and operations :

input :

print(2 and 3)

output :

3

input :

print(1 and 0)

output :

0

input :

print(1 and "I am doing good!")

output :

I am doing good!

input :

print([] and "Hello World!")

output :

[]

Here are some or operations :

input :

print(2 or 3)

output :

2

input :

print(None or 1)

output :

1

input :

print(0 or {})

output :

{}

input :

print([] or "Hello World!")

output :

Hello World!

Q: What are the values evaluated to False when applied to a Boolean operator?  
A:

* None and False.
* Zero of any numeric type: 0, 0.0, 0j.
* Empty sequences and collections: '', [], {}.
* Any remaining value is evaluated as True.

**The Strength of Strings in Python**

[Indexing&Slicing Strings](https://lms.clarusway.com/mod/lesson/view.php?id=11&pageid=28)

✔[String Formatting with Arithmetic Syntax](https://lms.clarusway.com/mod/lesson/view.php?id=11&pageid=29)

✔[String Formatting with '%' Operator](https://lms.clarusway.com/mod/lesson/view.php?id=11&pageid=31)

✔[String Formatting with 'string.format()' Method](https://lms.clarusway.com/mod/lesson/view.php?id=11&pageid=30)

✔[String Formatting with 'f-string'](https://lms.clarusway.com/mod/lesson/view.php?id=11&pageid=623)

Indexing&Slicing Strings

As we mentioned earlier, one of the most powerful aspects of Python is its string processing capability. 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, as follows:

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

**💡Tips:**

* 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

Let's see it in an example :

input :

city = 'Phoenix'

print(city[1:]) # starts from index 1 to the end

print(city[:6]) # starts from zero to 5th index

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

output :

hoenix

Phoeni

Ponx

hei

nix

xineohP

| **indexing_String** |
| --- |
| *Diagram of Indexing a String* |

You can use the len() function to find out the length (number of characters) of a text or a variable of any type.

input :

vegetable = 'Tomato'

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

output :

length of the word Tomato is : 6

**Q**: What is the output of print(str[4:]) if str = 'Python Language' ?  
**A**: on Language

**String Formatting with Arithmetic Syntax**

There are several ways in Python that we use when processing and using string data structures. The most important of these are:

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

We have stated to you what the **function** is? in the previous lessons. At this point, let us give the definition of the term **method**. 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. For example :

input :

print('clarus' + 'way')

output :

clarusway

We can also use \* operator for repeating the string without any spaces. For example :

input :

print(3\*'no way!')

output :

no way!no way!no way!

Examine the following example carefully :

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 👉🏻+=. Look at the examples below :

input :

fruit = 'orange'

fruit += ' apple'

print(fruit)

output :

orange apple

input :

fruit = 'orange'

fruit += ' apple'

fruit += ' banana'

fruit += ' apricot'

print(fruit)

output :

orange apple banana apricot

**Q**: There are several ways in Python that we use when processing and using string data structures. What are the most important of these:  
**A**:

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

**String Formatting with '%' Operator**

The other way that you will learn to format the strings is % operator. This one is not a frequently used way, but it's worth learning.

**' % '** operator formatting :

👉🏻% operator gets the values in order and prints them in order using several characters accordingly. Look at the example :

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.

**💡Tips:**

* 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. Here is an example :

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.

**Q**: What is the output of print('%.5s' % x) if x = "HelloWorld!" ?  
**A**: Hello

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

You can make strings change depending on the value of a variable or an expression. The main methods of Python to format the output are :

**'string.format()'** method :

string.format() method is the improved form of % operator formatting.

As in this example below, 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

**💡Tips:**

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

There is no limit in Python language! 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

**⚠️Avoid ! :**

* 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.

**String Formatting with 'f-string'**

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

**'f-string'** formatting :

It makes string formatting easier. This method was introduced in 2015 with Python 3.6.

**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}"**

Let's look at the example below on how the syntax is simple and readable.

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. Look at the examples:

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

There is also a multiline f-string formatting style. Follow the example :

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.

**Q**: If you want to use multiple 'f-string formatting' lines without parentheses, what will be the other option that you can use?  
**A**: You can use backslashes 👉\ between f-lines.

**Main String Operations**

Introduction

String processing is of great importance in the world of Data Science as a field of research and business in the world. Moreover, there seems to be much more to do in this field.

Good news! Python's string processing skills are very advanced that we will focus on some main parts of it.

A significant thing to keep in mind is that string is an immutable data type. This means you can't just change the string in place, so most string methods return a copy of the string. Well, how can we do that?

**💡Tips:**

* This is the way you should follow: You must create a new variable for the copy you made or assign the same name to the copy to save the changes made to the string for later use.

**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

These methods have optional arguments start and end. We can specify the search by adding arguments so that the area of search is delimited by start and end arguments.

**💡Tips:**

* Remember! Characters of string count from left to right and start with zero.

**The formula syntaxes are :**

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

Look at the example below:

input :

email = "clarusway@clarusway.com is my e-mail address"

print(email.startswith("@", 9))

print(email.endswith("-", 10, 32))

output :

True

True

**Q**: What are the string.startswith() and string.endswith() method used for? Describe how?  
**A**: 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.

**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 :

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

**Q**: print("Actions speaks louder than words".upper().swapcase().capitalize()), will this code work? If yes, what the output will be? Describe how?  
**A**: Yes it works. The syntax is : string.method(). Changing the string using these methods returns string type again. The output is :  
Actions speaks louder than words  
  
Follow the additional examples below :  
  
string.upper() # returns string type,  
string.upper().lower() # also returns string type,  
string.upper().lower().title() # returns string type again.

**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.

Now see the examples about how we implement these methods? :

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\_string.lstrip("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 :

interoperabili

As we said before, Python's string processing capability is very good. In this context, many other string processing and editing methods are available.

**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.

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

That is our first list in this course. Now let's print the list.

input :

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

print(country)

output :

['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New Zealand']

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Create this list with [**scratch**](https://scratch.mit.edu/projects/341599722/editor/).

**💡Tips:**

* 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. Let's create another list using list() function and compare with 👉🏻'[]'.

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']

**💡Tips:**

* 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]]**

**💡Tips:**

* As you see above, one or more of the **list elements** can even be a list.

Basic Operations with Lists

In Python, there are many methods and functions for dealing with the list structures. You'll learn some of them which are basic and the most common. Let's begin:

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']

| **append** |
| --- |
| *Diagram of '.append( )' Method* |

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

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']

| **Tiobe_Index** |
| --- |
| *Diagram of '.insert( )' Method* |

We can remove the elements in lists using list.remove() method or sort the elements using list.sort() method. Examine the example :

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']

**💡Tips:**

* Remember! Elements of a list are counted from left to right and start with zero as in string types.

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

**✏️Homework:**

* Guess and figure out the output of this syntax :

my\_list = [1, 3, 5, 7]

print(my\_list \* 3)

Show the Answer

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']

**✏️Homework:**

* Examine the use of index(), del() and pop() functions.

There are many other 'list operations' (mutable sequence types operations) methods [**here**](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types) that you can examine in detail.

**Accessing Lists**

**Introduction**

You know that there are several types of collections for storing data in Python, like **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. Now, let's see how can we do that?

**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**.

**⚠️Avoid ! :**

* Do not start indexing with **1**. The first index of the element of a list is **0**. We will never stop remembering that!

First, let's begin with a simple example :

input :

colors = ['red', 'purple', 'blue', 'yellow', 'green']

print(colors[2]) # If we start at zero,

# the second element will be 'blue'.

output :

blue

Now, let's learn the subject in detail through the examples :

input :

city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']

city\_list = []

city\_list.append(city) # we have created a nested list

print(city\_list)

output :

[['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

**city\_list** includes only **one** element which is the city list.

**💡Tips :**

* If you notice that city\_list has double square brackets.

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])

output :

Istanbul

'**city\_list[0][2]**' is a string type data. So, we can also access its elements via indexing. Let's access its third element :

input :

city\_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

print(city\_list[0][2][3])

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. Look at the example :

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]

**💡Tips :**

* Slicing is just similar to indexing. The difference is adding **colon** or **colons** in square brackets.

In slicing, pay attention to the stop index in square brackets. We got the elements from **index=2** to **index=5** (5 is not included): It means that we got 'second', 'third' and 'forth' element of the list.

You can keep in mind the formula syntax below for slicing a sequence. From '**start**' to '**stop-1**', by '**step**'.

**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.

Now let's apply this formula on a few examples. 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']

The following example slices the animals starts at **index=3** to the end.

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']

Q: In Python what is slicing?  
A: A mechanism to select a range of items from sequence types like list, tuple, strings etc. is known as slicing.

**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_index** |
| --- |
| *Diagram of Negative Indexing* |

**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']

| **negative_index_1** |
| --- |
| *Diagram-1 of Negative Slicing* |

input :

reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']

print(reef[:-3])

output :

['swordfish', 'shark', 'whale', 'jellyfish']

| **negative_index_2** |
| --- |
| *Diagram-2 of Negative Slicing* |

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']

| **negative_index_3** |
| --- |
| *Diagram-3 of Negative Slicing* |

input :

reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']

print(reef[::-2])

output :

['octopus', 'lobster', 'whale', 'swordfish']

**💡Tips :**

* If you choose negative step with the start and end indexes together, those should be used accordingly, that is, the end index should be less than the start index.

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]

[]

If you eager to find **more on lists** see [**here**](https://docs.python.org/3.8/tutorial/datastructures.html#more-on-lists).

**Q**: What does list[::-1] do?  
**A**: list[::-1] is used to reverse the order of a sequence of the elements in the list.

**Tuples**

**Definitions**

Up to this section of our lesson, we saw the most used collection types of Python : list. 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.

Q: What is the difference between list and tuple?  
A:  
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.

Let's create a simple empty tuple :

empty\_tuple = ()

This is our first tuple in this course. Now let's print its type.

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.

Let's examine some examples of creating tuples :

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

**✏️Homework:**

* Examine the use of .index() and .count() methods of the tuples.

**Benefits of Immutability**

Let's take a look at the basic advantages of tuples :

* 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.

If you want to go deep into tuples, you can find lots of information [**here**](https://docs.python.org/3.8/tutorial/datastructures.html#tuples-and-sequences).

**Dictionaries**

**Definitions**

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'}

**💡Tips:**

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

mix\_values = {'animal': ('dog', 'cat'), # tuple type

'planet': ['Neptun', 'Saturn', 'Jupiter'], # list type

'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}

**⚠️Avoid ! :**

* 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.

**Q**: What is a dictionary in Python?  
**A**: Python dictionary is one of the supported data types in Python. It is an **unordered collection of elements**. The elements in dictionaries are stored as key–value pairs. Dictionaries are indexed by keys. For example, below we have a dict named my\_dict. It contains two keys, **fruit** and **vegatable**, along with their corresponding values, **banana** and **onion**.  
  
my\_dict = {'fruit':'banana', 'vegatable':'onion'}

**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']**.

See the example.

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.

Look at the example :

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

| **nested_dictionary** |
| --- |
| *Diagram of Nested Dictionary* |

**💡Tips:**

* Dictionaries strongly resemble JSON syntax. The native json module in the Python standard library can be used to convert between JSON and dictionaries.

**✏️Homework:**

* What is '**JSON**' and what is it used for?

If you want to go deep into dicts, [**here**](https://docs.python.org/3.8/tutorial/datastructures.html#dictionaries) you will find what you want.

**Sets**

Definitions

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.

**⚠️Avoid ! :**

* 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.

Let's create a simple empty set :

empty\_set = set()

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

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']

**✏️Homework:**

* {'carnation', 'orchid', 'rose', 'violet'} 👈👉 {'rose', 'orchid', 'rose', 'violet', 'carnation'} Do these two sets give the same output and why? (Note: Try to figure out the answer before run on the Playground)

**Q**: Which one of the following is not the correct syntax for creating a set in Python?  
**A**:  
**a. set([[1,2],[3,4],[4,5]])**  
b. set([1,2,2,3,4,5])  
c. {1,2,3,4}  
d. set((1,2,3,4))  
  
**Explanation:** The iterable argument given for the set must be used in a correct way.

**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.(kesişim)
* **.union()** : Returns the unification of two sets.(birleşim)
* **.difference()** : Gets the difference of two sets.

Now, let's do some examples of these methods :

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.

Thus, we have completed this topic which is the most important one in Python.

**Conditional Statements**

[**Structure of the 'if' Statements**](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=55)

**✔**[**Comparison Operators**](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=56)

**✔**[**'if-else' Statements**](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=57)

**✔**[**'if-elif-else' Statements**](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=58)

**✔**[**Nested 'if-elif-else' Statements**](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=59)

**Structure of the 'if' Statements**

In some cases, your program needs to execute some part of the code only if a specific condition is true. The simple structure of an if statement is :

if condition:

body # the body code should start at indentation (four spaces)

The if statements check the condition. The condition is always a Boolean expression, that is, its value equals either True or False.

If it evaluates to True, it executes the body of the if statement. If it evaluates to False, it skips the body. This logic works like the English language.

input :

if True:

print('it is true')

output :

it is true

input :

empty\_seat = 14

if empty\_seat > 3: # in this case, 14>3=True, so the body will execute

print('there is still seat to sit')

output :

there is still seat to sit

💡Tips :

* Note that the condition ends with a **colon** (iki nokta üstüste) and a new line starts with an indentation.(satır başı)

**Comparison Operators**

Boolean values basically make it clear if a piece of code needs to be executed. Because comparisons result in bool, it's always best to use them as a condition.

Therefore, it is time to examine comparison operators :

| comparison_operators |
| --- |
| *Comparison Operators* |

| Operator | How it works ? | Sample |
| --- | --- | --- |
| == | Returns True if two values are equal or False if different | 2 == 2 (True), 2 == 3 (False) |
| != | Returns True if two values are not equal or False if equal | 2 != 2 (False), 2 != 3 (True) |
| > | Returns True if the value on the left is greater than the value on the right otherwise returns False | 3 > 2 (True), 2 > 3 (False) |
| < | Returns True if the value on the left is less than the value on the right otherwise returns False | 2 < 3 (True), 3 < 2 (False) |
| >= | Returns True if the value on the left is greater than or equal to the value on the right otherwise returns False | 3 >= 2 (True), 3 >= 3 (True), 2 >= 3 (False) |
| <= | Returns True if the value on the left is less than or equal to the value on the right otherwise returns False | 3 <= 2 (False), 3 <= 3 (True), 2 <= 3 (True) |

⚠️Avoid ! :

* **Do not use assignment operator = in comparison statements. In comparisons, you have to use == operator for equality.**

Now, let's learn the subject through an example :

input :

x = 6

y = 9

print ("is x equal to y? :" , x == y)

print ("is x not equal to y? :" , x != y)

print ("is x less than y? :" , x < y)

print ("is x greater than y? :" , x > y)

print ("is x less than or equal to y? :" , x <= y)

print ("is x greater than or equal to y? :" , x >= y)

output :

is x equal to y? : False

is x not equal to y? : True

is x less than y?? : True

is x greater than y? : False

is x less than or equal to y? : True

is x greater than or equal to y? : False

**'if-else' Statements**

In this part of the topic, we will first examine if-else statements.

An if-else statement is another kind of conditional statements in Python. It is used with an additional keyword: else.

else works like an if statement. If none of the conditions in if are ensured, "else" will be used to specify all remaining conditions. The simple structure of an if-else looks like :

if condition1:

execute body1

else : # if condition1 is not ensured execute body2

execute body2

💡Tips :

* Note that, else doesn't require any condition and the body2 requires 4-space indentation here.

With a few simple examples, you can be sure you will understand this topic much better.

input :

course = 'clarusway'

if course == "clarusway":

print("you guaranteed the job")

else:

print("think about it again")

output :

you guaranteed the job

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-Scratch Time ! : Solve this example with [scratch](https://scratch.mit.edu/projects/350671406/editor/).

input :

number = 5

if number <= 3:

print("Number is smaller than or equal to 3")

else: # Optional clause (you can only have one else)

print("Number is bigger than 3")

output :

Number is bigger than 3

**'if-elif-else' Statements**

The elif statement is used when it requires to specify several conditions in our program.

In Python you can deﬁne a series of conditionals using :

* if for the **ﬁrst** one,
* elif for the **rest**, up until the ﬁnal (optional),
* else for **anything not caught by the other conditionals**.

The basic structure of these statements looks like :

if condition1:

execute body1

elif condition2:

execute body2

else:

execute body3

How does the structure of statement work? : condition1 is checked first, and if the result is True the body1 in the 'if' statement' will be executed. If not, condition2 in the 'elif' statement is checked. If the result is True, body2 will be executed, if not, body3 in the last control point 'else' will be executed.

Let's take a look at the example below:

input :

weight = 80

if weight > 100:

print("That's too heavy!")

elif weight > 75:

print("I can lift that!")

else:

print("That's too light!")

output :

I can lift that!

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Solve this example with [**scratch**](https://scratch.mit.edu/projects/340876151/editor/).

We can use as many elif statements as we need, so your conditions can be varied.

| **if statements** |
| --- |
| *Diagram of General 'if-elif-else' Statement* |

As we stated before, the code inside the else block is executed only if all conditions before it is False. Let's see it in an example :

input :

audience = "baby"

if audience == "kid":

print("it is free to go to cinema")

elif audience == "teen":

print("discounted price!")

elif audience == "adult":

print("normal price")

else:

print("No such audience, stay at your home!")

output :

No such audience, stay at your home!

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Solve this example with [**scratch**](https://scratch.mit.edu/projects/341440492/editor/).

In this program, we grouped the prices of movie tickets based on the age ranges of the audience going to the cinema. If you pay attention, after the True or False check-in each level, the flow continues according to the response received. In the last step, else can be reached if all the answers are False. See the flow diagram below :

| **logic_flow** |
| --- |
| *Diagram of Logic Flow* |

**Nested 'if-elif-else' Statements**

Both if-else and if-elif statements can be nested. Let's see the nested structure on the same movie ticket example.

input :

audience\_group = 'kid', 'teen', 'adult'

audience = "teen"

if audience in audience\_group:

if audience == "kid":

print("it is free to go to cinema")

elif audience == "teen":

print("discounted price!")

else: # audience == "adult":

print("normal price")

else:

print("No such audience, stay at your home!")

output :

discounted price!

Let us write a program that asks you to enter your exam score and calculates the range in which your degree is based on your exam score. Try to run this code on your *Jupyter Lab* cell if available.

score = int (input("Enter your score :"))

if score >= 90:

if score >= 95:

Score\_letter="A+"

else:

Score\_letter="A"

elif score >= 80:

if score >= 85:

Score\_letter="B+"

else:

Score\_letter="B"

else:

Score\_letter="below B"

print ("Your degree: %s" % Score\_letter)

**💡Tips :**

* input() is a function that takes a value from the user and assigns it to a variable that you choose. We will use it very common.

Nested if-elif-else structures may seem a bit complicated to you, the best way to overcome this is to examine and practice plenty of samples.

If you eager to find **more on control flow tools** see [**here**](https://docs.python.org/3.8/tutorial/controlflow.html?highlight=else#more-control-flow-tools).

**Loops**

**Definitions**

When writing programs in Python, in some cases it is not enough to execute our block of code only once. The loops are used to repeat (iterate) (yinelemek) the execution of a block of code.

As one of the most main functions in programming, loops are an important part of almost every programming language. Loops enable programmers to set certain sections of their code to repeat through a number of loops which are referred to as iterations.

This topic covers using multiple types of loops and applications of loops in Python. You will learn two types of loops which are :

* **while Loop,**
* **for Loop.**

Q: What are the two major loop statements?  
A: for and while loops.

**'while' Loop**

while loops have a boolean logic, similar to if statements. As long as the result of the condition returns True, the code block under while loop runs. When the condition returns to False, the loop execution is terminated and the program control moves further to the next operation. Here is the simple structure of a while loop :

while condition:

body

| while_loop |
| --- |
| *Diagram of Basic Structure of while Loop* |

We will not use this loop as often as the for loop, but it is still worth understanding.

Let's create our first while loop :

input :

number = 0

while number < 6:

print(number)

number += 1

print('now, number is bigger or equal to 6')

output :

0

1

2

3

4

5

now, number is bigger or equal to 6

The variable *number* acts as a counter in this loop. This variable changes its value after each iteration. When the value of a counter reaches 6, the program control moves to the next operation and prints the text above.

⚠️Avoid ! :

* If we make a logical mistake in the loop variable (since you don’t increase your variable, a condition never becomes False and can work forever), we can start an infinite loop! For this reason, we have to specify a condition for the loop to give False to exit the loop.

We can call a list in while loop. Let's see an example :

input :

my\_list=["a", "b", "c", "d", "e"]

a = 0

while a < len(my\_list):

print('square of {} is : {}'.format(a, a\*\*2))

a+=1

output :

square of 0 is : 0

square of 1 is : 1

square of 2 is : 4

square of 3 is : 9

square of 4 is : 16

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-Scratch Time ! : Solve this example with [scratch](https://scratch.mit.edu/projects/341441228/editor/).

💡Tips :

* Did you remember? variable += number is the same as variable = variable + number.

Always use valid syntax and make comments. In the beginning, it may seem that the while loop is not so easy to apply, but several times later, you’ll understand that it’s a very useful tool. Lastly, let's play famous 'guessing a number game' using while loop :

answer = 44

question = 'What number am I thinking of? '

print ("Let's play the guessing game!")

while True:

guess = int(input(question))

if guess < answer:

print('Little higher')

elif guess > answer:

print('Little lower')

else: # guess == answer

print('Are you a MINDREADER!!!')

break

In the example above;

* We have written a program that does not exit the while loop until you find the correct number,
* We used break keyword in order to quit and exit the while loop,
* When the user knows the answer (44) and enters input, it takes the value of 44 and assigns to variable guess, in the end, else works and breaks the loop.

☝ Discuss it in-class!

If available, run this code on your *VS Code with JupyterNotebook* cell and try to understand how it works. Enjoy

**'for' Loop**

You'll learn one of the most used, very simple and useful syntaxes in Python: for loop. When you want to iterate a block of code you will use for loop. To create a for loop, you need a **variable** and an **iterable object**. Here is the simple structure of a for loop :

for variable in iterable :

code block

Let's examine the subject through an example. We need an iterable so we can use a list.

input :

for i in [1, 2, 3, 4, 5] :

print(i)

output :

1

2

3

4

5

You can follow the iterating steps of for-loop below :

| **Tiobe_Index** |
| --- |
| *How 'for-Loop' Works?* |

In the structure of the for loop, you can use also an iterable variable of course. See the example below :

input :

seasons = ['spring', 'summer', 'autumn', 'winter']

for season in seasons :

print(season)

output :

spring

summer

autumn

winter

**✏️Homework:**

* for i in {'n1' : 'one', 'n2' : 'two'} : print(i) Will this line of loop work? If **yes** what will be the output? If **no**, what is the problem? (Note: Try to guess the result or do research about this code, before running on the Playground or on your Jupyter Lab.

Q: How does for loop and while loop differ in Python and when do you choose to use them?  
A: **For loop** is generally used to iterate through the elements of various collection types such as list, tuple, set and dictionary.  
  
**While loop** is the actual looping feature that is used in any other programming language. This is how Python differs in handling loops from the other programming languages.

**Working with the Iterators**

Let us explain the term **iteration** a little more.

**Iterable** object can be anything for which items are received one by one, forward only. In Python, the process of recurrent execution of a block of code is called an **iteration**.

We can basically classify iterations as two headings :

* If the number of repetitions is predetermined, it is called **definite** iteration.
* The repetition structure that makes the code block run as long as the predetermined condition generates True is called **indefinite** iteration.

| **Loop** |
| --- |
| *Diagram of the Iterating Loop* |

For example; string, list, tuple, dictionary or set are the iterable types of data.

Let's use a string variable as an iterator in for loop :

input :

course = 'clarusway'

for i in course :

print(i)

output :

c

l

a

r

u

s

w

a

y

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Solve this example with [**scratch**](https://scratch.mit.edu/projects/341441774/editor/).

**Q**: What are Python iterators?  
**A**: Iterators in Python are array-like objects which allow moving on the next element. We use them in traversing a loop, for example, in a for loop.

**Operations with the 'for' Loop**

In this topic, you will learn about how we use the for loop using several functions and methods.

In the example below, you'll get a number from the user and print a sentence the number of times we receive from the user.

times = int(input("How many times should I say 'I love you'"))

for i in range(times):

print('I love you')

As we stated before, input() function can get the value of different data types and assign a variable you chose. In the example above, it gets a number and assigns it to times. If the user enters 3 then the output will be :

I love you

I love you

I love you

Now, let's write a code that asks the user a number between 1 and 10 and puts that number into the **multiplication table**.

n = int(input('enter a number between 1-10'))

for i in range(11):

print('{}x{} = '.format(n, i), n\*i)

☝ Discuss it in-class!

**✏️Homework:**

* Write the same code above by yourself but using 👉🏻**%** operator in print() function.

**💡Tips :**

* If you want the user to input numbers, use the input() function together with the int() function. Otherwise, the value entered by the user will be in the **string** data type.

Let's get to know the features of range() function in details:

The range() function creates an iterable sequence of numbers. And it can be simply converted into an iterable object: list, set, and tuple. For example :

input :

b = list(range(11))

print(b)

output :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

input :

a = set(range(0,10))

print(a)

output :

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

input :

c = tuple(range(11))

print(c)

output :

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

You can keep in mind the formula syntax below for range() function:

**The formula syntax is : range(start, stop, step)**

Besides, you can use starred expression 👉🏻**\*** before range() function to separate its elements. See these examples :

input :

print(range(5)) # it will not print the numbers in sequence

print(\*range(5)) # '\*' separates its elements

output :

range(0, 5)

0 1 2 3 4

input :

print(\*range(5,25,2))

output :

5 7 9 11 13 15 17 19 21 23

Starred expression 👉🏻**\*** can separate other **iterable** objects. For example, you can separate a string:

input :

print(\*('separate'))

output :

s e p a r a t e

☝ Discuss it in-class!

You can create reverse sequence numbers using a negative step.

input :

print(\*range(10,0,-2))

output :

10 8 6 4 2

In some cases, you will need to set up the for loop with multiple variables and the iterables. Examine the example :

input :

text = ['one','two','three','four','five']

numbers = [1, 2, 3, 4, 5]

for x, y in zip(text, numbers):

print(x, ':', y)

output :

one : 1

two : 2

three : 3

four : 4

five : 5

☝ Discuss it in-class!

**💡Tips :**

* zip() function make an iterator that aggregates elements from each of the iterables.

**Nested 'for' Loop**

As a programmer, you may sometimes need to interact with a single element of an iterable data and all other elements simultaneously, that is, your code block in a loop can also contain a loop. Yes, we're talking about nested loops.

In Python, you can easily place one loop inside another one. First outer loop then inner one runs. You'll see it in the following example :

input :

who = ['I am ', 'You are ']

mood = ['happy', 'confident']

for i in who:

for ii in mood:

print(i + ii)

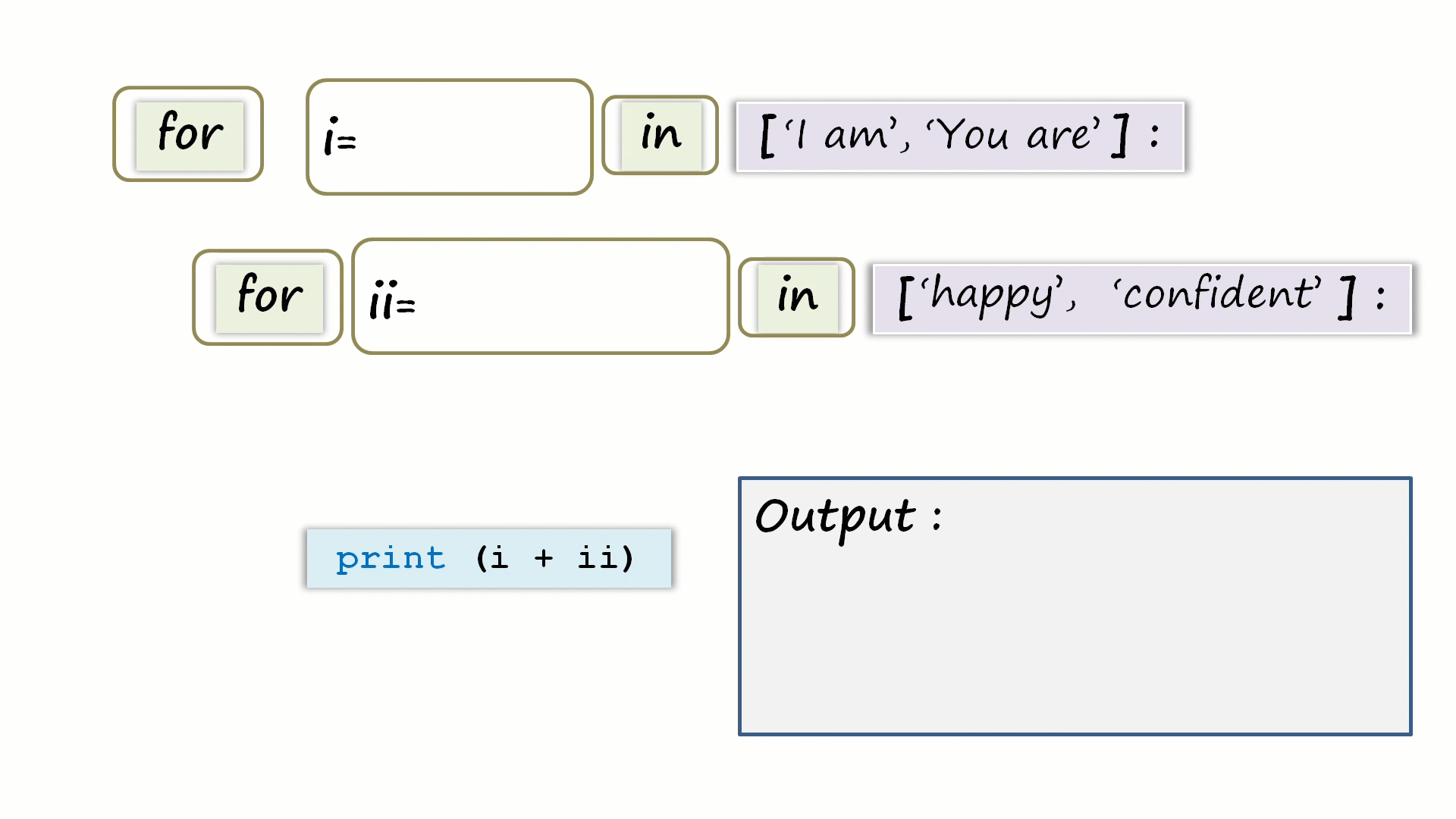
output :

I am happy

I am confident

You are happy

You are confident

You can better understand how it works, by watching the following animation.

| **Nested_For_Loop** |
| --- |
| *How 'nested-for-Loop' Works?* |

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Solve this example with [**scratch**](https://scratch.mit.edu/projects/341442714/editor/).

If you want to get deeper into it, you can find the details about the **loops**[**here**](https://docs.python.org/3.8/tutorial/controlflow.html?highlight=else#break-and-continue-statements-and-else-clauses-on-loops).

**FUNCTIONS**

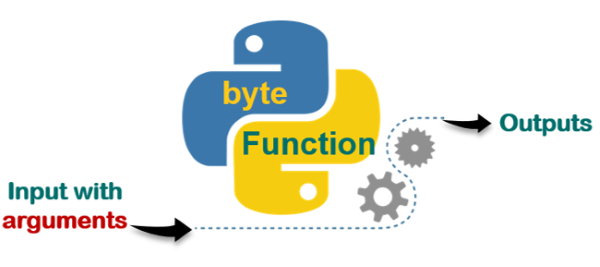
**Acquaintance with Functions**

Introduction

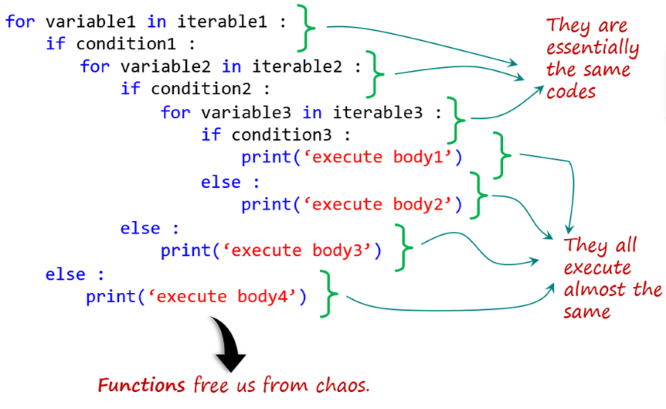
Basically, a **function** is a block of code that executes some logic for you, e.g. prints a text, deletes some data or square a number. In other words, a **function**is a piece of code that only runs when it is called.

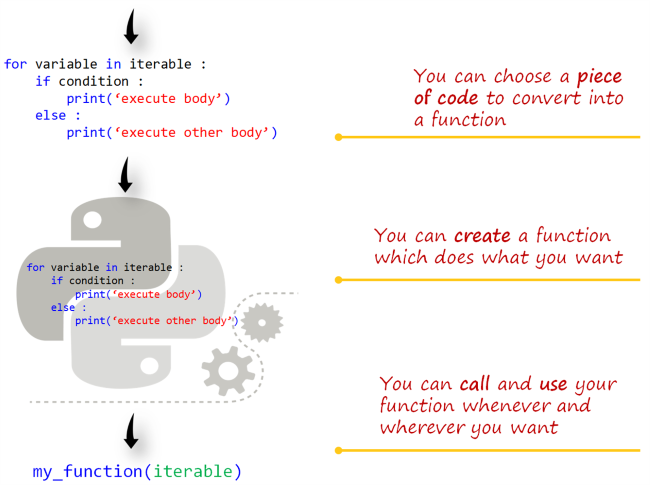
Functions in Python provide organized, reusable and modular code to perform a set of speciﬁc actions. Functions simplify the coding process, prevent redundant logic, and make the code easier to follow.

You can enter or input data, known as **arguments**, into a function and it returns/outputs something good that you want.



In some cases, you may need to create your own function. So that they help eliminate mess in your code because it saves you from unnecessary repetitions. You can **call**and use your function whenever and wherever you want. Follow a sample of flow diagrams of the functionalization process below. You can see below that we have several nested loops and conditional statements. As you noticed there are many repetitions of codes which look a bit messy and hard to understand.





Here we have tried to show you just the logical process of creating a function. As an example, my\_function is the name of the function that we have created. We have chosen a code block which consists of a **for-loop** and an **if-statement** to create a function. You will learn how we create and call a function with full syntaxes in the next lesson. Before **defining** (or creating) a function, let's take a look at how we **call** and use it.

**Q**: What is function in Python?  
**A**: A function is a block of code which is executed only when it is called. To define a Python function, the **def** keyword is used.

- **Interview Q&A**

**Calling a Function**

**Calling a Function Means Using It**

In the previous lesson, you have learned the basic philosophy of what a function is. In this lesson, we will examine how we **call** a function or what does 'calling a function' mean.

Reading a function is very easy in Python. For example; multiply(2, 5) or multiply(no1, no2). In this example, **multiply** is the name of the function, 2 and 5 are its **arguments** that we passed into the **parameters** which are the variables (no1 and no2). You can simply grasp that this function multiplies two numbers.

**💡Tips:**

* In fact, according to relevant [**Python documents**](https://docs.python.org/3/tutorial/controlflow.html#defining-functions), there is no significant difference between the definition and use of **parameter** and **argument** terms in Python. But, they are slightly different from each other.

Actually, **calling** a function means **using** it. When you need a function in your codes, you can simply use it. For example, if you want to multiply two numbers, you can just write the name of that function and the numbers (arguments) inside the parenthesis. By doing this, you have actually called this function. Look at the examples below :

input :

a = 3

b = 5

multiply(3, 5)

output :

15

input :

a = 3

b = 5

multiply(a, b)

output :

15

**! Avoid:**

* Here, don't try to run these examples on the Playground otherwise it gives an error. Remember, we haven't defined this function yet.

**Calling print() Function**

You have already used print() function dozens of times since the beginning of this course. You have also learned the details of print() function in the **Python Basics** course.  You are now very familiar with it. Nevertheless, let's take a look at what is what.  
  
In fact, what we do is solely writing its name and adding parentheses after it to call the print() function in your code. That's it!  
  
For example, let's consider this code : print("Say : I love you!") Here, in the example, you see the name of the function (print) followed by a sentence in parentheses. We can say that the sentence (Say: I love you!) that you passed into the () is an **argument**. We have a wide range of freedom of movement here. We may use the print() function with no argument besides we can also use it even with multiple arguments :

input :

print('Say: I love you!')

print()

print('me too', 2019)

output :

Say: I love you!

me too 2019

As you can see the outputs of the example above, we called that function (print()) three times. The first call printed a string, the second call printed an empty line and the third call printed two arguments which consist of one string and one integer data.

At this point, we advise you to examine [this function](https://docs.python.org/3/library/functions.html#print) again.

**Built-in Functions**

Frankly, our intention for this lesson is to inform you of the existence of the 'built-in' functions and to make you familiar with it. We are going to take a short tour of the 'built-in' functions. Besides, throughout the course, we will examine a significant part of the 'built-in' functions under some headings.

If you are considering a function which may do something that you want, it probably exists. You just need to be aware of its existence.

There are a range of functions and types built into the Python interpreter, so they are always usable. By the way, you don't have to worry about the term **interpreter**. We will talk about what it is in the next lessons.

In the latest version Python 3.9 the number of [**built-in functions**](https://docs.python.org/3/library/functions.html#built-in-functions) is 69. So far you have learned and used almost a dozen of these functions for various purposes. Such as : print(), int(), list(), input(), range().

These built-in functions are indeed very useful. They solve most of your needs without having to fall back on elsewhere.

It is a great benefit to have a quick look at the official Python documentation for the built-in functions mentioned below.

* Some of them return bool type according to the conditional algorithm in it. For example; all(*iterable*), any(*iterable*), and callable(*object*).
* Some of them help you convert data types into each other. For example; bool(), float(), int(), and str().
* Some others allow you to create and process the collection types. Such as : dict(), list(), tuple(), set(), len(), frozenset(), zip(), filter(*function*, *iterable*), and enumerate(*iterable*).
* Some others tackle numbers. Such as : max(), min(), sum(), and round().
* The others are built for special purposes. They do some complicated implementations. For example : map(*function*, *iterable*, *...*), eval(*expression*[, *globals*[, *locals*]]), sorted(*iterable*), open(), dir([*object*]), hash(), and help([*object*]).

In conclusion, we had a short tour of built-in functions and there is far more to explore, but don't ever give up!

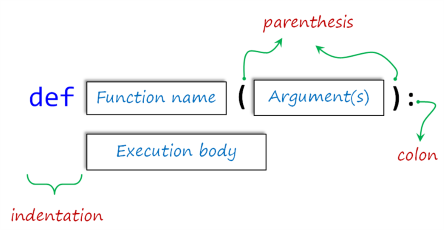
**Q**: Explain Python functions.  
**A**: A function is a section of the program or a block of code that is written once and can be executed whenever required in the program. A function is a block of self-contained statements which has a valid name, parameters list, and body. Functions make programming more functional and modular to perform modular tasks. Python provides several built-in functions to complete tasks and also allows a user to create new functions as well. There are two types of functions: Built-In Functions: copy(), len(), count() are the some built-in functions. User-defined Functions: Functions which are defined by a user known as user-defined functions.

- **Interview Q&A**

**Defining (Creating) a Function**

**Introduction**

Sometimes when you writing a program, built-in functions maybe not enough for you. Or sometimes you may need to use a block of code in your program repeatedly. Then you can write your own function. That is called the **user-defined function**in Python. Let's take a brief look at how it works.



The keyword **def** introduces the name of the function. It must be followed by the function name and the parenthesized list of formal arguments. The statements that form the body of the function start at the next line and must be indented (leave four spaces).

**Main Principles of 'Defining'**

The basic **formula syntax** of user-defined function is :

def function\_name(arguments) :

execution body

We define a Python function using the **def** statement, providing a name (so as to call it later) for our function and specifying either an empty or multiple argument list within parentheses. The rules for naming variables also apply here. So they should be written in **lowercase with underscores between words**. Argument lists are optional, but the parentheses are NOT. A colon 👉🏻**:** follows the closing parenthesis and indicates the start of our functions execution body. The function’s codes (execution body) MUST be indented under the **def** statement.

**⚠️Avoid:**

* Remember to put colon 👉🏻: just after the parentheses.
* Remember to leave four-space indentation at the beginning of the execution body lines.

Let's grasp the matter with an example :

def first\_function(argument\_1, argument\_2) :

print(argument\_1\*\*2 + argument\_2\*\*2)

This function, which we defined, gives the sum of the squares of arguments. Let's call and use it.

input :

first\_function(2, 3) # here, the values (2 and 3) are allocated to the arguments

output :

13

In the example above, the values (2 and 3) are allocated to the arguments provided at the function call in parentheses.

**💡Tips:**

* When there is**no indentation**, it means that the **definition process** of the function must **end**.

And now, let's define the multiplying function (multiply(a, b)) that you have seen as an example in the previous lesson.

input :

def multiply(a, b) :

print(a \* b)

multiply(3, 5)

multiply(-1, 2.5)

multiply('amazing ', 3) # it's really amazing, right?

output :

15

-2.5

amazing amazing amazing

As we have already stated, we can define a function without using any arguments. Let's give an example by leaving the parentheses empty.

input :

def motto() :

print("Don't hesitate to reinvent yourself!")

motto() # it takes no argument

output :

Don't hesitate to reinvent yourself!

If you want to go deeper, [here](https://docs.python.org/3/tutorial/controlflow.html#defining-functions)what you looking for.

**Q**: How do we write a function in Python?  
**A**: We can create a Python function in the following manner.  
Step-1: to begin the function, start writing with the keyword def and then mention the function name.  
Step-2: We can now pass the arguments and enclose them using the parentheses. A colon, in the end, marks the end of the function header.  
Step-3: After pressing an enter, we can add the desired Python statements for execution.

- **Interview Q&A**

**Execution of a Function**

The functions you have seen so far did not **return**any types or values but executed some actions. In order to use later the output and data types generated by the functions in our program flow, we need to define our function using the keyword return in addition to def. Let's see what happens in the following example :

input :

def multiply\_1(a, b) :

print(a \* b) # it prints something

def multiply\_2(a, b) :

return a \* b # returns any numeric data type value

multiply\_1(10, 5)

print(multiply\_2(10, 5))

output :

50

50

As you noticed, the outputs are the same. Then what is the difference? Well, the first function just prints some data that you passed into. The second one generates a numeric type value. If you check their types you will see :

input :

print(type(multiply\_1(10, 2)))

print(type(multiply\_2(10, 5)))

output :

20

<class 'NoneType'>

<class 'int'>

So, when we need it in our program, we can't use the result of the first function since it is **NoneType** data. But, the second one is **integer data** that we can use it in the future when we need it. Let's take a look at this subject using Python's best-known function.

input :

shadow\_var = print("It can't be assigned to any variable")

print(shadow\_var) # NoneType value can't be used

output :

It can't be assigned to any variable

None

In the example above, we can't assign the result of print() function to a variable.

**💡Tips:**

* Note that, if there are more than one keyword return in a function, then the execution of that function will end after the first return.

**Q**: What is the return keyword used for in Python?  
**A**: The purpose of a function is to receive the inputs and return some output. The return is a Python statement which we can use in a function for sending a value back to its caller.

- **Interview Q&A**

**Arguments vs Parameters**

We are aware that it may seem a bit difficult to understand clearly what **arguments**are. Likewise, the term **parameters**used in the same way as the arguments may seem also hard to grasp.

In fact, **arguments**are some kind of variable. So you can think of them as aliases for variables. This is exactly what we call the **parameters**. That is, the values ​​you assign to the parameters defined in a function are arguments.

If you look at the Python [documentation](https://docs.python.org/3/tutorial/controlflow.html?highlight=built%20function#more-on-defining-functions)on this topic, you will notice that the terms **parameters**and **arguments**are used almost the same way. But they are different things. You will better understand this topic through an example :

input :

def who(first, last) : # 'first' and 'last' are the parameters(or variables)

print('Your first name is :', first)

print('Your last name is :', last)

who('Guido', 'van Rossum') # 'Guido' and 'van Rossum' are the arguments

print()

who('Marry', 'Bold') # 'Marry' and 'Bold' are also the arguments

output :

Your first name is : Guido

Your last name is : van Rossum

Your first name is : Marry

Your last name is : Bold

As you can see, this function can be executed with different arguments. You can pass any two strings into the parameters.  
  
Since the function given in the example above takes two arguments, we need to pass exactly two arguments into it. If you passed only one argument into the parameter of the  who() function, it gives an error. Consider the following example :

input :

who('Joseph') # we passed only one argument into the function

output :

Traceback (most recent call last):

File "code.py", line 5, in <module>

who('Joseph')

TypeError: who() missing 1 required positional argument: 'last'

**⚠️Avoid:**

* Be careful. The **order** of **parameters**in the function must match the **order** of **arguments** you passed into.

There is much more to talk about the arguments.

**Correct Use of Arguments**

We will stick to relevant [Python documents](https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions) to make you understand this subject clearly. When calling a function, there are several ways to use arguments.

**Positional Arguments**

Actually, the arguments that you learned in the previous lesson is the positional one. Positions (sequence) of the arguments matter. When calling a function with **positional arguments**, they must be passed **in order from left to right**.

Take a look at these additional examples :

input :

def pos\_args(a, b):

print(a, 'is the first argument')

print(b, 'is the second argument')

pos\_args(3,4)

print()

pos\_args(4,3)

output :

3 is the first argument

4 is the second argument

4 is the first argument

3 is the second argument

input :

pos\_args('first','second')

print()

pos\_args('second', 'first')

output :

first is the first argument

second is the second argument

second is the first argument

first is the second argument

**Keyword Arguments**

If you do not want to allow the sequences/positions of the arguments to restrict you when you call a function, you can also call these arguments by keywords. Commonly and traditionally, **kwargs**is used as an abbreviation of **keyword arguments**.

The formula syntax is : kwargs=values.

Consider the following example :

input :

def who(first, last) : # same structure as the previous one

print('Your first name is :', first)

print('Your last name is :', last)

who(first='Guido', last='van Rossum') # calling the function is different

# we used kwargs to pass the values into the function

output :

Your first name is : Guido

Your last name is : van Rossum

 Consider the example taken from the official Python doc. :

def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):

print("-- This parrot wouldn't", action, end=' ')

print("if you put", voltage, "volts through it.")

print("-- Lovely plumage, the", type)

print("-- It's", state, "!")

accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

parrot(1000) # 1 positional argument

parrot(voltage=1000) # 1 keyword argument

parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments

parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments

parrot('a million', 'bereft of life', 'jump') # 3 positional arguments

parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword

**💡Tips:**

* If you have noticed the positions of the parameters voltage and action, sequences or positions don't matter when using **keyword arguments**.

To examine how it works, you'd better try the functions above one by one on the Playground.

Considering the defined functions, all the following calls would be invalid:

parrot() # required argument missing

parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument

parrot(110, voltage=220) # duplicate value for the same argument

parrot(actor='John Cleese') # unknown keyword argument

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actor is not a valid argument for the parrot function), and their order is not important. This also includes non-optional arguments (e.g. parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:

input :

def function(a):

pass # actually, 'pass' does nothing. it just moves to the next line of code

function(0, a=0)

output :

Traceback (most recent call last):

File "code.py", line 4, in

function(0, a=0)

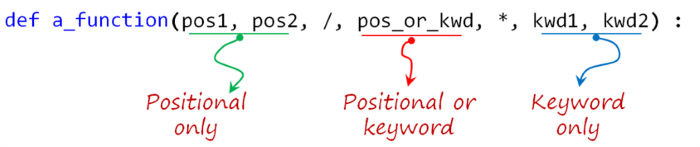
TypeError: function() got multiple values for argument 'a'

**Special Arguments (Optional)**

*--It'll be not included in the In-Class--*

According to the official related Python documents: By default, arguments may be passed to a Python function either by **position** or explicitly by **keyword**. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:



where 👉🏻/ and 👉🏻\* are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: *positional-only*, *positional-or-keyword*, and *keyword-only*. Keyword arguments are also referred to as *named arguments* or *named parameters.*

**Positional-or-Keyword Arguments**

If 👉🏻/ and 👉🏻\* are not present in the function definition, arguments may be passed to a function by position or by keyword. These are the arguments that we have already mentioned in the previous lesson.

Positional-Only Arguments

Looking at this in a bit more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters’ order matters and the parameters cannot be passed by keyword. Positional-only parameters are placed before a 👉🏻/ (forward-slash). The 👉🏻/ is used to logically separate the positional-only parameters from the rest of the parameters. If there is no 👉🏻/ in the function definition, there are no positional-only parameters.

Parameters following the 👉🏻/ may be *positional-or-keyword* or *keyword-only*.

Keyword-Only Arguments

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an 👉🏻\* in the arguments list just before the first *keyword-only* parameter.

**⚠️Avoid:**

* Do not confuse. When you use all kinds of arguments in a function definition, you have to pay attention the order of them.

**Arbitrary Number of Arguments**

**Default Arguments**

Before moving on to the subject of the **arbitrary number of arguments**, it is better to briefly mention the **default arguments**, which are one of the kinds of arguments.

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow.

When calling a function defined by **parameters with default values**, there is no obligation to pass any arguments into the function. Let's see how it works in an example :

input :

def city(capital, continent='Europe'):

print(capital, 'in', continent)

city('Athens') # we don't have to pass any arguments into 'continent'

city('Ulaanbaatar', continent='Asia') # we can change the default value by kwargs

city('Cape Town', 'Africa') # we can change the default value by positional args.

output :

Athens in Europe

Ulaanbaatar in Asia

Cape Town in Africa

As you see in the example, new values can be assigned to parameters either by name or by position. This kind of usage simplify the calling of a function, otherwise defining the function with default values of parameters makes no sense.

\*args and \*\*kwargs

The arguments we have used in the functions so far limit us to a certain extent. If we don't define the default values for the arguments, we will always need to pass the exact numbers of the arguments to match the number of the parameters defined in the function. But, there will be some situations when you might want to pass an arbitrary number of arguments.  
  
Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur. 

The formula syntax is : \*args.

Normally, these variadic arguments will be last in the list of formal parameters because they scoop up all remaining input arguments that are passed into the function. Any formal parameters which occur after the \*args parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

For example, let's define a function which takes two kinds of fruit and prints them.

input :

def fruiterer(fruit1, fruit2) :

print('I want to get', fruit1, 'and', fruit2)

fruiterer('orange', 'banana')

output :

I want to get orange and banana

What if the user wants to get more than two kinds of fruit? Since we don't know how many kinds of fruit each user will enter, using '**arbitrary numbers of arguments**' is the most intelligent method. Consider the following example :

input :

def fruiterer(\*fruit) :

print('I want to get :')

for i in fruit :

print('-', i)

fruiterer('orange', 'banana', 'melon', 'ananas')

output :

I want to get :

- orange

- banana

- melon

- ananas

As you can see above, we passed a list of fruits (arguments) into one parameter (fruit). Isn't it very useful?

If you need to prefer to use arbitrary keyword arguments (\*\*kwargs), you can use it in the same way.

The formula syntax is : \*\*kwargs.

You can examine the following example :

input :

def animals(\*\*kwargs):

for key, value in kwargs.items():

print(value, "are", key)

animals(Carnivores="Lions", Omnivores="Bears", Herbivores="Deers", Nomnivores="Human")

output :

Lions are Carnivores

Bears are Omnivores

Deers are Herbivores

Human are Nomnivores

As you can see in the example, in this type of argument (\*\*kwargs), we can determine the number of the arguments and their assigned value pairs by ourselves. In the next call of this function, we can use different arguments both in number and value from the above argument pairs.

**💡Tips:**

* Traditionally, people in the world of computer programming use \*args for the arbitrary number of positional arguments and \*\*kwargs for the arbitrary number of keyword arguments.

When calling a function defined by multiple positional parameters, using \*arg syntax in parentheses, we can pass all arguments into the function with a single variable. Likewise; When calling a function defined by multiple keyword arguments, using \*\*kwargs syntax in parentheses, we can pass all arguments which are in a dictionary form into the function with a single variable. Carefully examine the following examples :

input :

def brothers(bro1, bro2, bro3):

print('Here are the names of brothers :')

print(bro1, bro2, bro3, sep='\n')

family = ['tom', 'sue', 'tim']

brothers(\*family)

output :

Here are the names of brothers :

tom

sue

tim

input :

def gene(x, y): # defined by positional args

print(x, "belongs to Generation X")

print(y, "belongs to Generation Y")

dict\_gene = {'y' : "Marry", 'x' : "Fred"}

gene(\*\*dict\_gene) # we call the function by a single argument(variable)

output :

Fred belongs to Generation X

Marry belongs to Generation Y

input :

def gene(x='Solomon', y='David'): # defined by kwargs (default values assigned to x and y)

print(x, "belongs to Generation X")

print(y, "belongs to Generation Y")

dict\_gene = {'y' : "Marry", 'x' : "Fred"}

gene(\*\*dict\_gene)

output :

Fred belongs to Generation X

Marry belongs to Generation Y

Let's strengthen your knowledge of the 'arguments' thoroughly with more examples.

**Q**: What does this mean: \*args, \*\*kwargs? And why would we use it?  
**A**: We use \*args when we aren’t sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function. \*\*kwargs is used when we don’t know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments. The identifiers args and kwargs are a convention, you could also use \*bob and \*\*billy but that would not be wise. 

- **Interview Q&A**

**Recapitulation (Optional)**

*--It'll be not included in the In-Class--*

The use case will determine which parameters to use in the function definition:

def a\_function(pos1, pos2, /, pos\_or\_kwarg, \*, kwarg1, kwarg2, \*\*kwargs) :

As guidance:

* Use **positional-only** if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
* Use **keyword-only** when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
* Use **arbitrary numbers of arguments** (\*args) when you can't determine how many arguments your function needs. **\*args**enables you to have interoperability with a list of positional arguments in your function.
* You can use **\*\*kwargs**, when you don't know the exact number of keyword arguments in your function. **\*\*kwargs** enables you to have interoperability with a dictionary of key-value pairs.
* The order of the **parameters** you use when defining the function is as important as the order of the **arguments** you pass into when you call the function.

**✏️Homework:**

* Do a research on the definition and the usage of the **API**s.

**Scope of the Variables (Optional)**

Theoretical Definitions

*--It'll not be included in the In-Class--*

Let us give you some theoretical explanations about the term **namespace** and **scope** in Python. We will again stick to relevant [Python documents](https://docs.python.org/3/tutorial/classes.html?python-scopes-and-namespaces#python-scopes-and-namespaces) to make you understand this subject clearly. Besides, we think that examining these Python documents will be of great benefit to you.

**What is Namespace?**

A **namespace** is a system in which each **object** in Python has a *separate name*. An object could be a *method* or a *variable*. In other words, a **namespace** is a mapping from names to objects. Most namespaces are currently implemented as Python **dictionaries**, but that’s normally not noticeable in any way (except for performance), and it may change in the future.

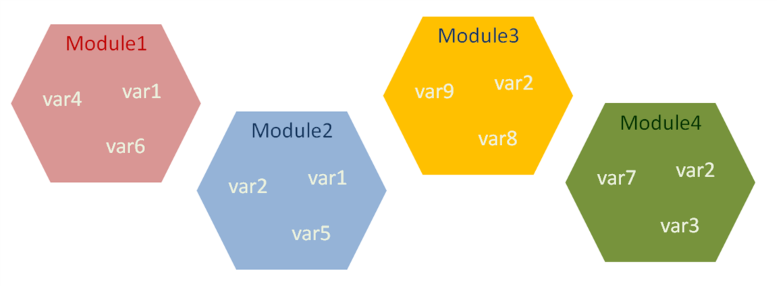
*For example*: imagine that you have two files named 'my\_python' on your computer both on **drive-C** and **drive-D**. You can easily access which file you want, by the file-path system. It can be easily understood which file is on which drive by looking at their file path.

In the program stream, the Python interpreter understands what specific method or variable one is trying to access in the code, depending on the namespaces.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up and is never deleted.

In the following figure, you can see some variables which have the same names are in the different *modules* (**namespaces**) at the same time. You can work with the variable that you want using this syntax : module.variable. Considering this figure, we can call var1 in Module2 like : module2.var1

You don't need to know what the module is and its definition for now. You will examine it in detail in the next lessons.



**What is Scope?**

A **scope** is a concept describes where or in which **space** the **variables** are defined in the program stream. This concept has a significant place in programming. In other words, a **scope** is a *textual region*of a Python program where a *namespace*is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

The term **scope** is mostly related to nested functions, modules, and the main program flow in accordance with the use of variables. It describes the accessibility and the existence of a variable.

A **scope** defines the **hierarchical order** in which the names of the variables have to exist in order to match **names** with the **objects**(variables).

Now, let's put all these definitions into practice with a simple example :

input :

my\_var = 'outer variable'

def func\_var():

my\_var= 'inner variable'

print(my\_var)

func\_var()

print(my\_var)

output :

inner variable

outer variable

As you can see in the example, the name of the variable (my\_var) has been used both in the function (func\_var) and at the top of the main program stream. When you call the function (func\_var) or print directly the variable (my\_var), you were probably noticed that the same variable produces different outputs. This is because of the location (space) of that variable, that is, where or in which space it is defined in the program flow.

After learning what the concept of scope theoretically is, let's examine the **global** and **local** variables.

**Q**: What is the namespace in Python?  
**A**: The namespace is a fundamental idea to structure and organize the code that is more useful in large projects. A namespace is defined as a simple system to control the names in a program. It ensures that names are unique and won't lead to any conflict. Also, Python implements namespaces in the form of dictionaries and maintains name-to-object mapping where names act as keys and the objects as values.

- **Interview Q&A**

**Global and Local Variables**

*--It'll not be included in the In-Class--*

When you define a variable in the Python program stream it is global or local, depending on in which space it is defined.

**Global Variable**

If the variable you define is at the highest level of a module, that variable becomes **global**. So you have the freedom to use this global **variable** in a block of code anywhere in your program.

Global variables allow us to make some interactions between functions. **For example,** suppose we store the credentials of a student who has applied for Clarusway in a **global variable**. Let's assume that we use this global variable many times in 3 **different functions** that we have defined regarding course activities. The **global variable** provides us with convenience when the credentials of the person change. Only when we rearrange the information in this global variable will our variables in all functions be rearranged.

**Local Variable**

The variables you have defined **in a function** body are **local**. The name of this variable is therefore **only valid** in the function body to which it is located.**Local variables** eliminate some of the confusion risks that global variables can cause.

Let's take a look at this example to grasp the difference between global and local variables:

input :

text = "I am the global one"

def global\_func():

print(text) # we can use 'text' in a function

# because it's a global variable

global\_func() # 'I am the global one' will be printed

print(text) # it can also be printed outside of the function

text = "The globals are valid everywhere "

global\_func() # we changed the value of 'text'

# 'The globals are valid everywhere' will be printed

def local\_func():

local\_text = "I am the local one"

print(local\_text) # local\_text is a local variable

local\_func() # 'I am the local one' will be printed as expected

print(local\_text) # NameError will be raised

# because we can't use local variable outside of its function

output :

I am the global one

I am the global one

The globals are valid everywhere

I am the local one

---------------------------------------------------------------------------

NameError: name 'local\_text' is not defined

In the above example, we have seen that a *global variable* can be accessed not only from the top-level of the module but also from the body of the function. On the other hand, a *local variable* is valid only in the function's body it is defined. So, it is accessible from inside the nearest scope level and can not be accessed from the outside.

**💡Tips:**

* You might have a question about where you will need to use these issues. But, if you are writing a relatively long algorithm, you will eventually need to work with the nested functions and modules.

**Q**: What are local variables and global variables in Python?  
**A**: Variables declared outside a function or in global space are called global variables. These variables can be accessed by any function in the program. Any variable declared inside a function is known as a local variable. This variable is present in the local space and not in the global space. When you try to access the local variable outside the function, it will give an error.

- **Interview Q&A**

**LEGB Ranking Rule**

*--It'll not be included in the In-Class--*

When you call an **object** (*method or variable*), the *interpreter*looks for its name in the following order:

1. **Locals**. The space which is searched first, contains the local names defined in a function body.
2. **Enclosing**. The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope (from inner to outer), contains non-local, but also non-global names.
3. **Globals**. It contains the current module’s global names. The variables defined at the top-level of its module.
4. **Built-in**. The outermost scope (searched last) is the namespace containing built-in names.

The order given above is known as **LEGB** Ranking Rule. Let's see how it works in an example :

input :

variable = "global"

def func\_outer():

variable = "enclosing outer local"

def func\_inner():

variable = "enclosing inner local"

def func\_local():

variable = "local"

print(variable)

func\_local()

func\_inner()

func\_outer() # prints 'local' defined in the innermost function

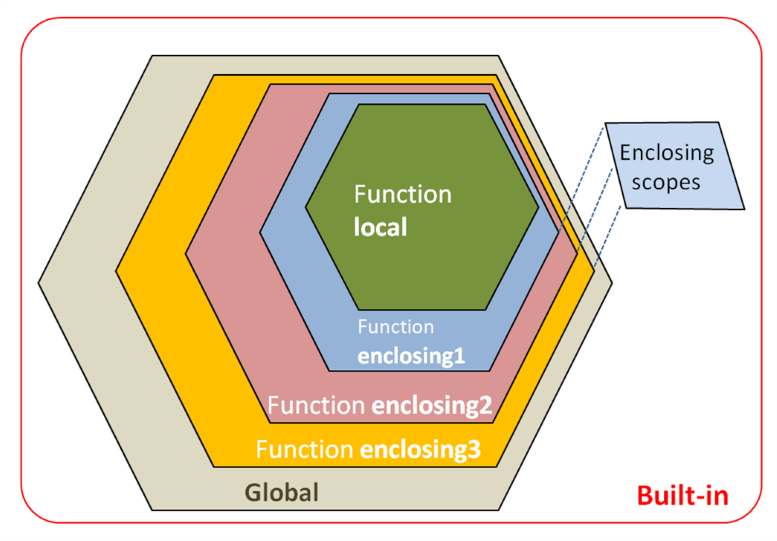
print(variable) # 'global' level variable holds its value

output :

local

global

In this example, during the execution of the code lines, the interpreter has to resolve the name '*variable*'.  The searching order of the variable names will be as follows : 'local' in func\_local, 'enclosing inner local' in func\_inner, 'enclosing outer local' in func\_outer, globals and built-in names. You can examine LEGB Rule in the following figure. 



**'global' and 'nonlocal'**

*--It'll not be included in the In-Class--*

You know from the previous lesson that a variable defined in a function body becomes local. In some cases, we want to work with the variables defined as a global scope in the function body. Normally they are perceived *globally* and processed accordingly.

Or we may need to work with the nonlocal variables in the function body. The keywords global and nonlocal save us from these restrictions.

**Keyword 'global'**

You can not change the value assigned to a globally defined variable within a function. To do this we use the keyword global. If you examine the example below you will understand better.

input :

count = 1

def print\_global():

print(count)

print\_global()

def counter():

print(count)

count += 1 # we're trying to change its value

print() # just empty line

counter()

output :

1

Traceback (most recent call last):

File "code.py", line 11, in <module>

counter()

File "code.py", line 8, in counter

print(count)

UnboundLocalError: local variable 'count' referenced before assignment

As you can see in the example above, if you try to assign a value **contains local variable expressions** to a **global variable** within a function, *UnboundLocalError* will raise. We've tried to assign a value to the count variable using an expression that contains the count variable. This is because the interpreter can't find this *variable* in the **local scope**. So, let's use the keyword global to solve this problem.

input :

count = 1

def counter():

global count # we've changed its scope

print(count) # it's global anymore

count += 1

counter()

counter()

counter()

output :

1

2

3

The reason for the error in the previous program is that the variable (count) we tried to modify could not be found by the interpreter in the local scope. It's because we used a *global variable* in the *local scope*.

**Keyword 'nonlocal'**

On the other hand, you can use the keyword nonlocal to extend the scope of the *local variable* to an upper scope. Consider the examples of nonlocalization :

input :

def func\_enclosing1():

x = 'outer variable'

def func\_enclosing2():

x = 'inner variable'

print("inner:", x)

func\_enclosing2()

print("outer:", x)

func\_enclosing1()

output :

inner: inner variable

outer: outer variable

We will make the variable x nonlocal so we can use its inner-value in the outer function (scope). Let's see.

input :

def enclosing\_func1():

x = 'outer variable'

def enclosing\_func2():

nonlocal x # its inner-value can be used in the outer scope

x = 'inner variable'

print("inner:", x)

enclosing\_func2()

print("outer:", x)

enclosing\_func1()

output :

inner: inner variable

outer: inner variable

**💡Tips:**

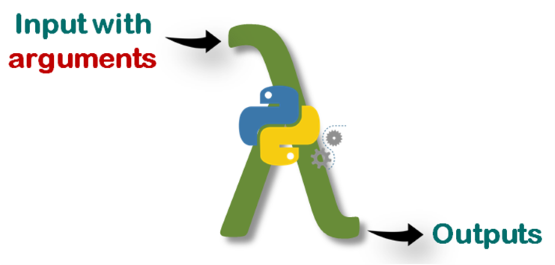
* Frankly, these keywords are not widely used in programming but are worth discussing.

**Lambda Functions**

**Defining a Lambda Function**

Another way to define functions in Python is lambda functions. Lambda functions are also called **anonymous**functions since they have no name. We use keyword lambda to define a function.

**The formula syntax is**: lambda parameters : expression



**Why we need lambda functions?**

If you need to use a one-time function, defining a lambda function is the best option. In some cases, you may need to define a function only once without having to use it later. For instance; let's square given numbers with a function. First, we're going to use def :

def square(x)

return x\*\*2

And now we'll define lambda function to do the same.

lambda x: x\*\*2

As you see, lambda is very simple and has a single line with a single expression. On the other hand, these two functions do exactly the same thing.

A lambda function can take multiple arguments separated by commas, but it **must** be defined with a *single expression*. This expression is evaluated and the result is returned

**⚠️Avoid:**

* Note that you do not need to use return statement in lambda functions.

Consider the following example of multiple arguments. Let's calculate the arithmetic mean of two numbers :

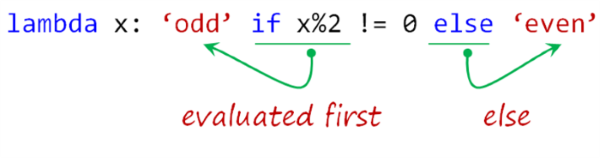
lambda x, y: (x+y)/2 # takes two numbers, returns the result

What if we need to use conditional statements within the lambda definition? Here how we do it :

lambda x: 'odd' if x % 2 != 0 else 'even'

**The formula syntax of conditional lambda statement is**:

lambda parameters : **first\_result** if conditional statement else **second\_result**



**⚠️Avoid:**

* Note that you can't use the [**usual conditional statement**](https://lms.clarusway.com/mod/lesson/view.php?id=18&pageid=57) with lambda definition.

If lambda is doing the same things with def then you might think of why and where do we use **lambda**? In the next lesson, we will try to find the answer to this question.

**Q**: What is a lambda function?  
**A**: A lambda function is an anonymous function (a function that does not have a name) in Python. To define anonymous functions, we use the ‘lambda’ keyword instead of the ‘def’ keyword, hence the name ‘lambda function’. Lambda functions can have any number of arguments but only one statement.

**Uses of the Lambda Functions**

So far you have seen the definition of lambda function and some of its features. Well, unlike def, where do we use lambda? If we need, how do we use the **lambda functions** in our code stream? Moreover, they don't even have names, so how can we call them? In this and the next lesson, we're going to try to find out the answer to these questions.

Lambda's most important advantages and uses are:

1. You can use it with its own syntax using ***parentheses***,
2. You can also *assign* it to a ***variable***,
3. You can use it in several ***built-in*** functions,
4. It can be useful inside **user-defined** functions (def).

Let's see how these work:

* **By enclosing the function in parentheses** :

(lambda x: x\*\*2)(2) # squares '2'

**The formula syntax is**: (lambda parameters : expression)(arguments)

Let's print the output :

input :

print((lambda x: x\*\*2)(2))

output :

4

Or you can use multiple arguments using the same syntax :

input :

print((lambda x, y: (x+y)/2)(3, 5)) # takes two int, returns mean of them

output :

4.0

You can also assign the lambda statement in parentheses to a variable :

input :

average = (lambda x, y: (x+y)/2)(3, 5)

print(average)

output :

4.0

* **By assigning a function object to a variable**:

Alternatively, you can assign the lambda function definition to a variable then you can call it :

input :

average = lambda x, y: (x+y)/2

print(average(3, 5)) # we call

output :

4.0

* You will see how we use lambda definition within some **built-in** or **user-defined**functions in the next lesson.

**Q**: What Are The Principal Differences Between The Lambda And Def?  
**A**:  
- Def can hold multiple expressions while lambda is a uni-expression function.  
- Def generates a function and designates a name to call it later. Lambda forms a function        object and returns it.  
- Def can have a return statement. Lambda can’t have return statements.   
- Lambda supports to get used inside a list and dictionary.

- **Interview Q&A**

**Lambda within Built-in (map()) Functions-1**

When using some built-in functions we may need additional functions inside them. This can be done by using def, but when we do the same thing with lambda we save both time and additional lines of code and we make it clear to read.

* **Lambda within map() function :**

map() returns a list of the outputs after applying the given function to each element of a given iterable object such as list, tuple, etc.

**The basic formula syntax is**: map(function, iterable)

Let's square all the numbers in the list using map() and lambda. Consider this example :

input :

iterable = [1, 2, 3, 4, 5]

map(lambda x:x\*\*2, iterable)

result = map(lambda x:x\*\*2, iterable)

print(type(result)) # it's a map type

print(list(result)) # we've converted it to list type to print

print(list(map(lambda x:x\*\*2, iterable))) # you can print directly

output :

<class 'map'>

[1, 4, 9, 16, 25]

[1, 4, 9, 16, 25]

☝ Discuss it in-class!

If you try to do the same thing using def, it is likely that the lines of code similar to the following occur. As you can see below, there are at least two additional lines of code. Moreover, we will not use the square function again because we only need to use it inside the map() function.

def square(n): # at least two additional lines of code

return n\*\*2

iterable = [1, 2, 3, 4, 5]

result = map(square, iterable)

print(list(result))

Now, let's try to give an example with multiple arguments in **lambda function** using map() :

input :

letter1 = ['o', 's', 't', 't']

letter2 = ['n', 'i', 'e', 'w']

letter3 = ['e', 'x', 'n', 'o']

numbers = map(lambda x, y, z: x+y+z, letter1, letter2, letter3)

print(list(numbers))

output :

['one', 'six', 'ten', 'two']

In the above example, we have combined three strings using 👉🏻**+** operator in the lambda definition. 

☝ Discuss it in-class!

**💡Tips :**

* Note that map() takes each element from iterable objects one by one and in order.

**Q**: What is map function in Python?  
**A**: map function executes the function given as the first argument on all the elements of the iterable given as the second argument. If the function given takes in more than 1 arguments, then many iterables are given.

- **Interview Q&A**

Lambda within Built-in (filter()) Functions-2

* **Lambda within filter() function :**

filter() filters the given sequence (iterable objects) with the help of a function (lambda) that tests each element in the sequence to be true or not.

**The basic formula syntax is**: filter(function, sequence)

Let's grasp the subject with an example in which we'll filter the even numbers in a list.

input :

first\_ten = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

even = filter(lambda x:x%2==0, first\_ten)

print(type(even)) # it's 'filter' type,

# in order to print the result,

# we'd better convert it into the list type

print('Even numbers are :', list(even))

output :

<class 'filter'>

Even numbers are : [0, 2, 4, 6, 8]

**💡Tips :**

* Note that filter() filters each element in the iterable object, depending on whether the function's result is True or False.

This time, we'll filter the vowels from the first ten letters in the list.

input :

vowel\_list = ['a', 'e', 'i', 'o', 'u']

first\_ten = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

vowels = filter(lambda x: True if x in vowel\_list else False, first\_ten)

print('Vowels are :', list(vowels))

output :

Vowels are : ['a', 'e', 'i']

☝ Discuss it in-class!

We draw your attention to this issue that *lambda definition* we use in this example gives only True or False as a result.

**Lambda within User-Defined Functions**

* **Lambda within def :**

Using a lambda statement in a user-defined function provides us useful opportunities. We can define a group of functions that we may use later in our program flow. Take a look at the following example :

input :

def modular\_function(n):

return lambda x: x \*\* n

power\_of\_2 = modular\_function(2) # first sub-function derived from def

power\_of\_3 = modular\_function(3) # second sub-function derived from def

power\_of\_4 = modular\_function(4) # third sub-function derived from def

print(power\_of\_2(2)) # 2 to the power of 2

print(power\_of\_3(2)) # 2 to the power of 3

print(power\_of\_4(2)) # 2 to the power of 4

output :

4

8

16

The modular\_function takes one argument, number ***n***, and returns a function that takes the power of any given number ***x*** by that ***n***.

This usage enabled us to use a function as flexible. Thanks to lambda, we could use a single def in different ways with the arguments we wanted. We've created three sub-functions derived from a single def. This is flexibility!

We can define a function with the same logic as the previous example that repeats the string passed into it.

input :

def repeater(n):

return lambda x: x \* n

repeat\_2\_times = repeater(2) # repeats 2 times

repeat\_3\_times = repeater(3) # repeats 3 times

repeat\_4\_times = repeater(4) # repeats 4 times

print(repeat\_2\_times('alex '))

print(repeat\_3\_times('lara '))

print(repeat\_4\_times('linda '))

output :

alex alex

lara lara lara

linda linda linda linda