General Information about Python

What is Python?

As you can imagine, Python is a programming language (such as Java, C ++, R, Ruby, and so on). Like other programming languages, it allows you to control the machine in front of you, the computer.

The IT industry is booming with Data science applications using,

* Artificial intelligence,
* Deep learning and
* Machine learning algorithms.

Python is the most widely used technologies in this domain. With the new-age applications, demand for a Python developer has also increased.

According to the [Google search trends](https://trends.google.com/trends/explore?q=%2Fm%2F05z1_,%2Fm%2F07sbkfb,%2Fm%2F02p97,%2Fm%2F0jgqg), in 2019, Python is the most popular searched term, among all programming languages.

According to Tiobe.com's index of programming languages, Python is the fastest-growing language.

| **Tiobe_Index** |
| --- |
| *Tiobe's 2019 Index of Programming Languages* |

Python has become the second-most-popular language in GitHub, overtaking Java for the first time, according to GitHub’s 2019 State of the [Octoverse report](https://octoverse.github.com/) on the usage of the popular code-sharing site.

| **GitHub_Index** |
| --- |
| *GitHub's 2019 Index of Programming Languages* |

Programs can be developed very quickly with this language. In addition, the simple and clean syntax of the Python programming language has made it a preferred language by many programmers. It's easy to write programs and read a program written by others. So, it has been widely used - especially in Data Science - and has received lots of demands in recent years.

| **Python Usage** |
| --- |
| *Some Python Usage Areas* |

**✏️Homework:**

* Do a research on which world-class companies build their technical infrastructure using Python programming language.

**Q**: What is Python?  
**A**: Python is a programming language. It allows you to control the computer. The benefits of Pythons are that it is simple and easy, portable, extensible.

**- Interview Q&A**

## General Information about Python

### The Programming Language of the Agile Era (Optional)

The contents in this section are developed by making use of the official brochure (Python Brochure Vol.I 2nd Edition, named: Case Studies & Success Stories), which was prepared by the **Python Software Foundation** (PSF) to introduce Python. You can access all the brochure [**here**](https://brochure.getpython.info/).

#### **Programming with Python**

Software quality is a vital ingredient to success in industry and science. Ubiquitous IT systems control the business processes of the global economy. Increasingly powerful computers and sophisticated algorithms provide the platform for new scientific discoveries. And global communication is inconceivable without intelligent software. In the race for customers, the pole position belongs to those who get to market faster than their competitors. Better and more creative solutions combined with the ability to respond instantly to new challenges drive the race. Writing secure and reliable programs in a fraction of the time normally required gets you first across the finish line.

**The Programming Language of the Agile Era :**

Agility is the hallmark of our times and Python is the programming language of the agile era. The Python universal programming language is the turbocharger of the IT department. Compared with other modern programming languages such as Java or C, Python achieves superior results in significantly shorter timescales for a number of different reasons.

For example, Python is a very lean programming language. Python programs are a great deal shorter than code written in other modern programming languages. As a result, both development times and maintenance costs are drastically reduced. Less code means fewer errors, meaning the cost of identifying and eliminating these errors is also reduced.

A comprehensive standard library and thousands of additional libraries in the Python Package Index provide developers with high-quality solutions that they can easily integrate into their applications to meet virtually any requirement.

In this way, Python frees up vast resources, which can be earmarked for more productive use elsewhere.

**The Master Key for System Integration :**

Python offers unique benefits for system integration. On the one hand, there are huge numbers of Python libraries, with which virtually any third-party system can be integrated. On the other hand, the libraries of many other programming languages can also be used in Python.

Once they have been programmed, Python applications can run on all operating systems for which a Python interpreter exists, significantly reducing the cost of operating-system-specific applications.

**The Language that has Changed Everything :**

For over 20 years, Python has been used successfully throughout the world as a programming language in industry, in the service sector, and also in research and science to meet a wide range of different requirements. In this time, the language has changed many things.

The Python programming language is easy to learn. It has blurred the boundaries between users and developers. Increasing numbers of scientists, engineers, financial experts, and others with little programming experience are using Python to solve specific complex technical problems.

## General Information about Python

### Historical Development of Python

This programming language was developed in the early 90s by a Dutch programmer called Guido van Rossum. Most people think that this programming language is named after the python snake, assuming its name is Python.

However, contrary to the assumption, the name of this programming language does not come from the python snake. Guido van Rossum named this programming language inspired by the show of **Monty Python’s Flying Circus** from an English comedy group called **The Monty Python**.

| **Rossum** |
| --- |
| Guido van Rossum |

Even though this is the case, the **logo** of the Python programming language with a serpent type in many places has become almost a tradition.



This language has a huge group of developers around the world. If you have any problem, you can always ask other Python users/developers for help or find a suitable answer on several sites like [**stackoverflow.com**](https://stackoverflow.com/)

General Information about Python

Review of Tools & Installations

The version of Python 3.X was released in 2008. It makes Python more *readable* and *consistent* than previous versions. Throughout this course, we will use the latest versions of **Python 3.X** (currently Python 3.9.0).

There are several Integrated Development Environments (IDE) that we can write and run Python codes on them. We prefer **Anaconda Navigator (Anaconda3) package** program which includes several IDEs options for you. You can **optionally** install Anaconda on your computer. Currently, Python 3.7 is available in the Anaconda package.

Among the IDEs in Anaconda, we prefer to use **Jupyter Lab** to write down the Python codes on it. The Jupyter Lab is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. **Jupyter Notebook** can also be used which is very similar to Jupyter Lab.

Apart from Anaconda, of course, **Python Shell** (IDLE) can also be used as a basic interpreter tool for Python.

**Throughout this course**;

* We will provide you **Playground** module to write and run your code on each lesson page. Therefore, as we stated before **you don't have to** install Anaconda Package, it is **optional**.
* Additionally, we will be using all Jupyter Notebook based IDEs, including the Google Colab application, during in-class sessions.
* You will have several **Scratch** exercises in accordance with Python codes. You can find an explanatory video below about how to solve the Scratch exercises.

If you decide to install Anaconda, you can visit the installation and user guide [**here**](https://lms.clarusway.com/course/view.php?id=7&section=1).

Here is an alternative way to run your code. This alternative program is **Visual Studio Code (VS Code)** which is also available in the Anaconda Package. Although we prefer to use *Jupyter Lab* in Anaconda Package to run our codes, you can install and use the VS Code program either.

VS Code will be used in the **interactive complementary video lessons** you will receive throughout the course. You can watch the video below about the installation of Python and VS Code if you want.

* **How to Get Started with VS Code**

You can also watch a video below about some configuration issues of VS Code.

* **Configuring of VS Code**

## First Steps into Coding

### Introduction

In this lesson, we will show you how to write and develop a code outline with Python programming language. Although these initial codes are quite simple, they will give you great tips on what Python language is like.

### First Program for 'Being a Good Person'

Let's write our first program. We will print 'being a good person' on the screen:

input :



1

2

print('being a good person')

output :



1

2

being a good person

As you see, it prints what you have written between single quotes like: 'text'. Here is another single line of code:

input :



1

2

print('clarusway will change my life')

output :



1

2

clarusway will change my life

https://drive.google.com/uc?export=view&id=1AAtBbH3F0uSt3rVDf7PhBS7L1VcMQAh-**Scratch Time ! :**Solve this example with [**scratch**](https://scratch.mit.edu/projects/341646035/editor/).

We wrote clarusway will change my life using print() **function**. For now, you do not need to understand how this code runs, just start to enjoy the coding looking like the English language.

If we make some statements at this point, print is the name of a function. A **function** is a piece of code that executes some logic for you, e.g. prints a text or square a number.

In some cases, a function is a subprogram we produce which can be reused within your programs. If the name of a function is followed by parentheses, it means that it was called to get the result.

**💡Tips:**

* Surrounding the expression with triple quotes: """...""" or '''...''' ensures that the code returns no error, especially in long texts.

## First Steps into Coding

### Matter of Quotes

As a Python programmer, you will deal with quotes a lot. As you know, a string text is surrounded by a pair of quotes.

There are basically two types of quotes we use. **Single** or **double** quotes. Both are the same but we should use them in the correct way:

* Use double quotes if your string includes the single one:e.g. "It's my pleasure!"
* Use single quotes if your string includes the double one:e.g. 'He said: "I am done" and fell down.'
* Use triple quotes if your string is too long which composed of multiple lines : e.g. :

'''

...long string...

..long string..

'''

Here's an example that shows using double-quotes. Note that, the text (it's not a problem) below includes a single quote '.

input :



1

2

print("it's not a problem")

output :



1

2

it's not a problem

Here is a new single-line program too. In this case, be careful with double-quotes. It is the same as single quotes. Even triple quotes can be used for the same result. Try to guess output of this code : print('''it's not a problem using "triple" quotes''').

**⚠️Avoid ! :**

* Do not mix two quotes style in one string.

print('We should have enough time for our family") This code gives an error because it starts with single quotes ends with double. See below:

input :



1

2

print('We should have enough time for our family")

output :



1

2

3

4

5

File "", line 1

print('We should have enough time for our family")

^

SyntaxError: EOL while scanning string literal

Let's do the same with numbers:

input :



1

2

print(572)

output :



1

2

572

input :



1

2

print(3.14)

output :



1

2

3.14

input :



1

2

print('3.14')

output :



1

2

3.14

**💡Tips:**

* If you have noticed we used 👉🏻. not 👉🏻, for the decimal number: 3.14.
* Surrounding the expression with quotes makes it in string type. We will see immutable types (int, string, tuple, etc.) in the next lessons.

We assumed that you have learned how to write simple Python code consisting of a single line which just prints a text. But, the real-world codes of Python contain a huge number of lines. So, now we will write codes that print multiple lines.

Let's look at the example below. And this is our first multi-line code. The following code prints three lines:

input :



1

2

3

4

print('no pain')

print('no gain')

print(3.14)

output :



1

2

3

4

no pain

no gain

3.14

If you need an empty line, you can use only print() function. Let's try :

input :



1

2

3

4

print('first line')

print() # second line is empty

print('third line')

output :

PEP 8 Conventions

What is PEP 8 ?

**PEP** stands for Python Enhancement Proposal. PEP 8 is a coding convention, a set of recommendations, about how to write your Python code more readable.

In other words, PEP 8 is a document that gives coding conventions for the Python code comprising the standard library in the main Python distribution. One of Guido's (author of Python) key insights is that code is read much more often than it is written.

The guidelines **(PEP 8)** provided [**here**](https://www.python.org/dev/peps/pep-0008/#introduction) are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.

A style guide is about consistency. Consistency with this style guide is **important**. Consistency tutarlılık within a project is **more important**. Consistency within one module or function is the **most important**.

However, know when to be inconsistent. Sometimes style guide recommendations aren't just applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask someone else.

The main idea of PEP 8 is to use the same code style for all Python projects as if they were written by the same programmer. PEP 8, even for beginners, assures that it will easily understand the code written by any other developer.

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

## PEP 8 Conventions

### Some Important PEP 8 Rules

We will show you some important PEP 8 traditional rules that you can follow.

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

Immediately inside parentheses, brackets or braces :

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

Between a trailing comma and a following close parenthesis :

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

Immediately before a comma, semicolon, or colon :

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

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

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

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

| **YES** | **NO** |
| --- | --- |
| x = 3 | x =mmmmm3 |
| y = 4 | y =mmmmm4 |
| long\_vars = 5 | long\_vars = 5 |

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

Failure to follow the basic rules of PEP 8 does not make your program wrong or unable to work. In the near future, you will learn a lot about Python and become a more skilled programmer, but it will always be important to follow the code style.

There's nothing to worry about following PEP 8. You don't need to learn the traditional PEP 8 rules all at once right away. When you need it, you can open and read it now and then. We will also show you some PEP 8 conventions throughout this course.

You can check if your code complies with traditional PEP 8 rules using this [**module**](http://pep8online.com/)**.**

## Comments and Docstrings

### Introduction

When writing a program, you will need to add explanatory notes to others or even yourself. The longer you write lines, the better you will understand the necessity of this. We can add these explanatory notes to our program as 'comment' or as 'docstring' in more detail.

## Comments and Docstrings

### Comments

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

There are three types of commenting methods. These are :

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

input :



1

2

# This is a single line comment

* **Inline comments** satir ici aciklamalar also begin with hash character 👉🏻# and start from the end of a code line.

input :



1

2

print('the cosmos has no superiority to chaos') # This is an

    inline comment

output :



1

2

the cosmos has no superiority to chaos

* **Multi-line comments** basically consist of multiple comment lines.

input :



1

2

3

4

5

print(3 + 4)

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

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

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

output :



1

2

7

**💡Tips:**

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

Apart from the well-readable syntax itself, in writing Python programs, there are other important things that contribute to understandability of your program. We assume that you are familiar with comments and how they help in understanding codes.

In the real programming world, comments become especially important as the program gets bigger and more complicated. Without using them, things may get confusing for other developers who see your code for the first time. It may get confusing even for you within a couple of months after writing the program.

**⚠️Avoid ! :**

* More comments don't necessarily need to be better. If code is self-explanatory, comments are unnecessary.
* Do not make unnecessary comments. Usually, comments should answer the question **why** as opposed to **what**.
* When necessary, **update your comment**. Be sure that your comments will not be in contradiction to the code.

Q: What are the comments and how do you write it in Python?  
A: Comments are used to explain code when the basic code itself isn't clear. Python ignores comments, and so will not execute code in there, or raise syntax errors for plain English sentences. Comments in Python start with a # character. '#' character converts all subsequent characters to the comment form that Python does nothing.  
  
# this is a single line comment  
  
print("Hello World!") # this is an inline comment

## Comments and Docstrings

### Docstrings

We have to say at the beginning that you will not learn to create and write docstrings in this course. Only what we will show you is what docstrings are and how we will call and display it.

**Docstrings** are - unlike regular comments - stored as an attribute of the function or the module they document, meaning that you can access them programmatically. Docstring runs as an explanatory text of codes and it should be written between triple quotes. Like: """docstring""".

**💡Tips:**

* You don't need to learn or know; 'what the **function** and the **module** are?' for now.
* We will show you these topics in the Python Basics Plus Course.

Although it is not mandatory to learn, for now, you can consider the definitions of these terms below:

* We have briefly mentioned its meaning before. A **function** is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions.
* A **module** is a Python object with arbitrarily named attributes that you can bind and reference. Simply, a module is a file consisting of Python code. A module can define functions, classes, and variables. A module can also include runnable code.

Normally, when we want to call docstring of a function or module to read, we will use \_\_doc\_\_ (the keyword doc enclosed by double underscores) syntax. See the example below :

Here is an example:

input :



1

2

3

4

5

6

7

8

9

def function(): # Don't be confused, we use 'def()' to create a

    function.

# You will see it in the next lessons.

"""

Hi, I am the docstring of this code.

If you need any information about this function or module, read me

    .

It can help you understand how the module or function works.

"""

print(function.\_\_doc\_\_)

output :



1

2

3

4

Hi, I am the docstring of this code.

If you need any information about this function or module, read me.

It can help you understand how the module or function works.

Let's see the docstring of print() function:

input :



1

2

print(print.\_\_doc\_\_)

output :



1

2

3

4

5

6

7

8

9

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys

    .stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Q: What is docstring in Python?  
A: Docstrings are - unlike regular comments - stored as an attribute of the function or the module they document, meaning that you can access them programmatically. Docstring runs as an explanatory text of codes and it should be written between triple quotes.

## Naming Variables

### General Description

As you know, each variable has a unique name that distinguishes it from others. Giving a good name to a variable may not be as simple as it sounds.

A Python **variable** is a reserved memory location to store values. In other words, a variable gives data to the computer for processing. We will discuss variables in detail in the next lessons.

**💡Tips:**

* Remember, a nice and meaningful naming of variables is a skill that can be gained over time. Of course, you also need to be familiar with PEP 8 traditional rules.

Expert programmers care much for naming the variables well to make their codes easy to understand. It is important because programmers spend a lot of time reading and understanding code written by other programmers.

The convention of naming is optional. You can use any names you like but it is useful to follow the convention so that someone (including you) knows what you have written.

## Naming Variables

### Conventional (PEP 8) Naming Rules

If variables have poor names, even your own code may seem unclear to you in a couple of months. Now let's learn how to choose good names for our variables in accordance with PEP 8 rules:

* Choose lowercase words and use underscore to split the words:
* price = [22, 44, 66],
* low\_price = 12.00
* Do not use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single-character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. If you want to use 'l', use 'L' instead.
* l = 'It is not correct use',
* O = "It's also incorrect use"

**⚠️Avoid ! :**

* Do not use specific Python keywords (name of a function or phrase) as a name, like sum, max, min, in, or, for, etc.
* Use a sensible name. The variable name needs to be legible and meaningful and explain to the reader what types of values will be stored in it.
* figures = 'this is better',
* f = 'it is not meaningful'
* Don't choose too common names. Use a name to describe the meaning of the variable. However, try to limit it to no more than 3 words.
* If the word you intend to choose is long, try to find the most common and expected short form to make it easy to predict later.

| **Variable to be named** | **Sample of Good name** | **Sample of Bad name** | **Why bad?** |
| --- | --- | --- | --- |
| Cleaned Data | cleaned\_data | cdat | it doesn't make sense enough. |
| Indexes of the Clear Application Syntaxes | clr\_app\_syntx | ix\_app\_syntax | it doesn't make sense enough. |
| Customer Information of the Bank Accounts | customer\_bank\_info | costomer\_info\_bank\_account | it's too wordy. |

Q: Which of the following is an invalid statement?  
A:  
a) x, y, z = 1, 22, 333  
b) x\_y\_z = 1\_234\_567  
c) xyz = 1234567  
**d) x y z = 111 222 333**  
  
Spaces are not allowed in variable names

Basic Data Types

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.

## Basic Data Types

### 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.

Basic Data Types

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.

## Basic Data Types

### 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.

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 :



1

2

3

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

print(type(example1))

output :



1

2

<class 'str'>

input :



1

2

3

example2 = '71'

print(type(example2))

output :



1

2

<class 'str'>

input :



1

2

3

example3 = 71

print(type(example3))

output :



1

2

<class 'int'>

input :



1

2

3

example4 = 71.0

print(type(example4))

output :



1

2

<class 'float'>

input :



1

2

3

example5 = 3.14j

print(type(example5))

output :



1

2

<class 'complex'>

input :



1

2

3

example6 = True

print(type(example6))

output :



1

2

<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:

**💡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.

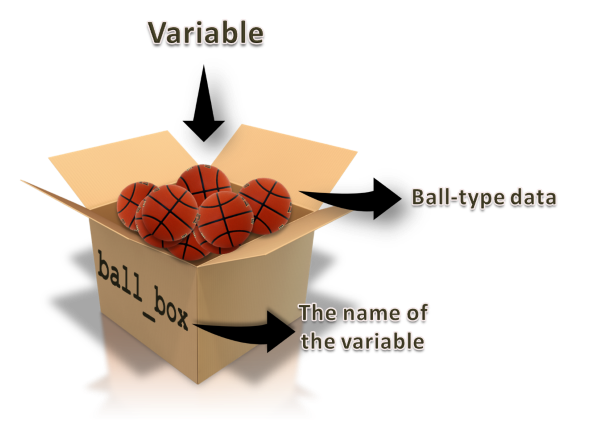
Basic Data Types

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 :



1

2

3

4

5

6

7

8

9

10

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 :



1

2

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:

**⚠️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

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 :



1

2

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

output :



1

2

15

input :



1

2

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

output :



1

2

40.0

input :



1

2

3

4

no1, no2 = 46, 52

no3 = no1 - no2

print(no3)

output :



1

2

-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 :



1

2

3

no1 = 46

print(no1/23) # division gives float

output :



1

2

2.0

input :



1

2

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

output :



1

2

6.0

input :



1

2

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

output :



1

2

3

input :



1

2

3

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

# it means 9 is an odd number

output :



1

2

1

input :



1

2

print(3\*\*2)

output :



1

2

9

input :



1

2

print(2\*\*3)

output :



1

2

8

input :



1

2

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

output :



1

2

8.0

input :



1

2

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

output :



1

2

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 : +, -

imple Operations

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 :



1

2

3

4

number = 2020

text = "children deserve respect as much as adults in"

print(text, number)

output :



1

2

children deserve respect as much as adults in 2020

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

input :



1

2

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

output :



1

2

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 :



1

2

3

4

5

print('i', end=' ')

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

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

print('to my mother')

output :



1

2

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 :



1

2

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

output :



1

2

smoking + is + slowly + killing me

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

input :



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

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 :



1

2

3

4

5

6

7

8

9

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 :



1

2

3

4

fruit = 'Orange'

vegetable = "Tomato"

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

output :



1

2

Orange and Tomato

**💡Tips:**

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

Simple Operations

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 :



1

2

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

output :



1

2

3

C:\november

umber\_expenditure.txt

input :



1

2

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

output :



1

2

one two three

input :



1

2

3

print('we', '\bare', '\bunited') # remember, normally print() function

# separates expressions by spaces

output :



1

2

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 :



1

2

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

output :



1

2

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

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**.

- Interview Q&A

Boolean Operations

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 | |

Boolean Operations

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 :



1

2

3

logic = True and False or not False or False

print(logic)

output :



1

2

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

- Interview Q&A

## Boolean Operations

### 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 :



1

2

print(2 and 3)

output :



1

2

3

input :



1

2

print(1 and 0)

output :



1

2

0

input :



1

2

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

output :



1

2

I am doing good!

input :



1

2

print([] and "Hello World!")

output :



1

2

[]

Here are some or operations :

input :



1

2

print(2 or 3)

output :



1

2

2

input :



1

2

print(None or 1)

output :



1

2

1

input :



1

2

print(0 or {})

output :



1

2

{}

input :



1

2

print([] or "Hello World!")

output :



1

2

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

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 :



1

2

3

4

5

6

7

8

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 :



1

2

3

4

5

6

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 :



1

2

3

4

5

6

7

8

9

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 :



1

2

3

4

5

6

7

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 :



1

2

3

4

vegetable = 'Tomato'

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

output :



1

2

length of the word Tomato is : 6

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

## The Strength of Strings in Python

### 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 :



1

2

print('clarus' + 'way')

output :



1

2

clarusway

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

input :



1

2

print(3\*'no way!')

output :



1

2

no way!no way!no way!

Examine the following example carefully :

input :



1

2

3

4

5

fruit = 'Orange'

vegetable = 'Tomato'

print("using + :", fruit + vegetable)

print("using \* :", 3 \* fruit)

output :



1

2

3

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 :



1

2

3

4

5

fruit = 'orange'

fruit += ' apple'

print(fruit)

output :



1

2

orange apple

input :



1

2

3

4

5

6

7

fruit = 'orange'

fruit += ' apple'

fruit += ' banana'

fruit += ' apricot'

print(fruit)

output :



1

2

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.

## The Strength of Strings in Python

### 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 :



1

2

3

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

print (phrase)

output :



1

2

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 :



1

2

3

4

sentence = "apologizing is a virtue"

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

output :



1

2

apologizing

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

input :



1

2

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

output :



1

2

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

## The Strength of Strings in Python

### 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 :



1

2

3

4

5

fruit = 'Orange'

vegetable = 'Tomato'

amount = 4

print('The amount of {} we bought is {} pounds'.format(fruit, amount))

output :



1

2

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 :



1

2

print('{state} is the most {adjective} state of the {country}'.format(state

    ='California', country='USA', adjective='crowded'))

output :



1

2

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 :



1

2

print('{0} is the most {adjective} state of the {country}'.format('California',

    country='USA', adjective='crowded'))

output :



1

2

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 :



1

2

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

    'found', 'I will'))

output :



1

2

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.

## The Strength of Strings in Python

### 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 :



1

2

3

4

5

6

7

fruit = 'Orange'

vegetable = 'Tomato'

amount = 6

output = f"The amount of {fruit} and {vegetable} we bought are totally {amount}

    pounds"

print(output)

output :



1

2

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 :



1

2

3

4

result = f"{4 \* 5}"

print(result)

output :



1

2

20

input :



1

2

3

4

5

my\_name = 'JOSEPH'

output = f"My name is {my\_name.capitalize()}"

print(output)

output :



1

2

My name is Joseph

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

input :



1

2

3

4

5

6

7

8

9

10

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 :



1

2

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 :



1

2

3

4

5

6

7

8

9

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 :



1

2

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.

Lists

You have earned 0 point(s) out of 0 point(s) thus far.

Introduction

There are various collection types in Python. While types such as int and str hold a single value, collection types hold multiple values.

In your programs, you usually need to group several items to render as a single object. We use collection types of data to do this job.

One of the most useful collections in Python is a list. In Python, a list is only an **ordered collection** of valid Python values.

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

### Creating a List

A list can be created by enclosing values, separated by commas, in square brackets 👉🏻[].

Let's create a simple list that includes some country names.



1

2

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

    Zealand']

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

input :



1

2

3

4

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

    Zealand']

print(country)

output :



1

2

['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 :



1

2

3

4

5

6

7

8

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 :



1

2

3

['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.

Lists

You have earned 0 point(s) out of 0 point(s) thus far.

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.



1

2

3

4

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 :



1

2

3

4

5

6

empty\_list\_1 = []

empty\_list\_1.append('114')

empty\_list\_1.append('plastic-free sea')

print(empty\_list\_1)

output :



1

2

['114', 'plastic-free sea']

input :



1

2

3

4

5

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

    'Sydney']

city.append('Addis Ababa')

print(city)

output :



1

2

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

    'Addis Ababa']

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

looks like list.insert(index, object). See the example :

input :



1

2

3

4

5

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

    , 'Addis Ababa']

city.insert(2, 'Stockholm')

print(city)

output :



1

2

['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 :



1

2

3

4

city = ['New York', 'London', 'Stockholm', 'Istanbul',

    'Seoul', 'Sydney', 'Addis Ababa']

city.remove('London')

print(city) # we have deleted 'London'

output :



1

2

['New York', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney',

    'Addis Ababa']

input :



1

2

3

4

city = ['New York', 'Stockholm', 'Istanbul', 'Seoul',

    'Sydney', 'Addis Ababa']

city.sort() # lists the items in alphabetical order

print(city)

output :



1

2

['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 :



1

2

3

city = ['Addis Ababa', 'Istanbul', 'New York', 'Seoul',

    'Stockholm', 'Sydney']

print(len(city))

output :



1

2

6

**✏️Homework:**

* Guess and figure out the output of this syntax :



1

2

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 :



1

2

3

4

city = ['New York', 'Stockholm', 'Istanbul', 'Seoul',

    'Sydney', 'Addis Ababa']

city[1] = 'Melbourne' # we assign 'Melbourne' to index 1

print(city)

output :



1

2

['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?

Accessing Lists

You have earned 0 point(s) out of 0 point(s) thus far.

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 :



1

2

3

4

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

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

# the second element will be 'blue'.

output :



1

2

blue

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

input :



1

2

3

4

5

6

7

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

    'Sydney']

city\_list = []

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

print(city\_list)

output :



1

2

[['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 :



1

2

3

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

    'Sydney']]

print(city\_list[0]) # access to first and only element

output :



1

2

['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 :



1

2

3

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

    'Sydney']]

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

output :



1

2

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 :



1

2

3

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

    'Sydney']]

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

output :



1

2

a

Accessing Lists

You have earned 0 point(s) out of 0 point(s) thus far.

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 :



1

2

3

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 :



1

2

[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 :



1

2

3

4

5

count = list(range(11))

print(count)

print(count[0:11:2])

output :



1

2

3

[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.

Accessing Lists

You have earned 0 point(s) out of 0 point(s) thus far.

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 :



1

2

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

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

input :



1

2

3

reef = ['swordfish', 'shark', 'whale', 'jellyfish',

    'lobster', 'squid', 'octopus']

print(reef[::-1]) # we have produced the reverse of the

    list

output :



1

2

['octopus', 'squid', 'lobster', 'jellyfish', 'whale',

    'shark', 'swordfish']

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

input :



1

2

3

reef = ['swordfish', 'shark', 'whale', 'jellyfish',

    'lobster', 'squid', 'octopus']

print(reef[::-2])

output :



1

2

['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 :



1

2

3

4

5

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 :



1

2

3

[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.

**- Interview Q&A**

Tuples

You have earned 0 point(s) out of 0 point(s) thus far.

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)

Tuples

You have earned 0 point(s) out of 0 point(s) thus far.

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 :

1

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')

Tuples

You have earned 0 point(s) out of 0 point(s) thus far.

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 :



1

2

3

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

    ----------------

TypeError: 'tuple' object does not support item assignment

**✏️Homework:**

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

You have earned 0 point(s) out of 0 point(s) thus far.

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).

Unfold (unco lapse) all regions

Toggle breakpoint

Dictionaries

You have earned 0 point(s) out of 0 point(s) thus far.

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.

## Dictionaries

You have earned 0 point(s) out of 0 point(s) thus far.

### 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'}

**- Interview Q&A**

Dictionaries

You have earned 0 point(s) out of 0 point(s) thus far.

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

Dictionaries

You have earned 0 point(s) out of 0 point(s) thus far.

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 :

2

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.

Sets

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.
* **.union()** : Returns the unification of two sets.
* **.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.

#### **After finishing the task correctly, then submit your answer (code) as plain text which shows you did correctly.**

**Task :**Find out the most frequent number and its frequency.

Write a program that;

* + Finds out the most frequent number in the given list.
  + Calculates its frequency.
  + Prints out the result such as :

| **Example** | |
| --- | --- |
| **Given list** | **Desired Output** |
| numbers = [1, 3, 7, 4, 3, 0, 3, 6, 3] | the most frequent number is 3 and it was 4 times repeated |

***Note****: You can/should use useful/necessary****built-in functions****and****methods****of the*list*operation.*

* [Assignment-009/2 (Comfortable Words)](https://lms.clarusway.com/mod/assign/view.php?id=10453)

**💡Objective:**

* + To improve your **knowledge** of **collection**types (set), **boolean**logic and raise your awareness of **"ten fingers keyboard"**.

#### **After finishing the task correctly, then submit your answer (code) as plain text which shows you did correctly.**

**Task :**Find out if the given word is "**comfortable words**" in relation to the *ten-finger keyboard use*.

* + A *comfortable word* is a word which you can type always alternating the hand you type with (assuming you type using a Q-keyboard and use of the ten-fingers standard).
  + The word will always be a string consisting of only letters from a to z.
  + Write a program which returns True if it's a comfortable word or False otherwise.

| **Examples** | |
| --- | --- |
| **Given  word** | **Desired Output (explanation)** |
| tester | False (uses only left-hand fingers)\* |
| polly | False (uses only right-hand fingers)\* |
| clarusway | True (uses both hand fingers)\* |

***Note****: Do a quick research on ten-fingers keyboard usage. (\*) the explanation doesn't need to be in the output.*

A comfortable word is a word which you can type always alternating the hand you type with (assuming you type using a QWERTY keyboard and use fingers as shown in the image below).

That being said, create a function which receives a word and returns true/True if it's a comfortable word and false/False otherwise.

The word will always be a string consisting of only ascii letters from a to z.

To avoid problems with image availability, here's the lists of letters for each hand:

Left: q, w, e, r, t, a, s, d, f, g, z, x, c, v, b

Right: y, u, i, o, p, h, j, k, l, n, m

Conditional Statements

You have earned 0 point(s) out of 0 point(s) thus far.

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 and a new line starts with an indentation.

### 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* |

| **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 te 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 :

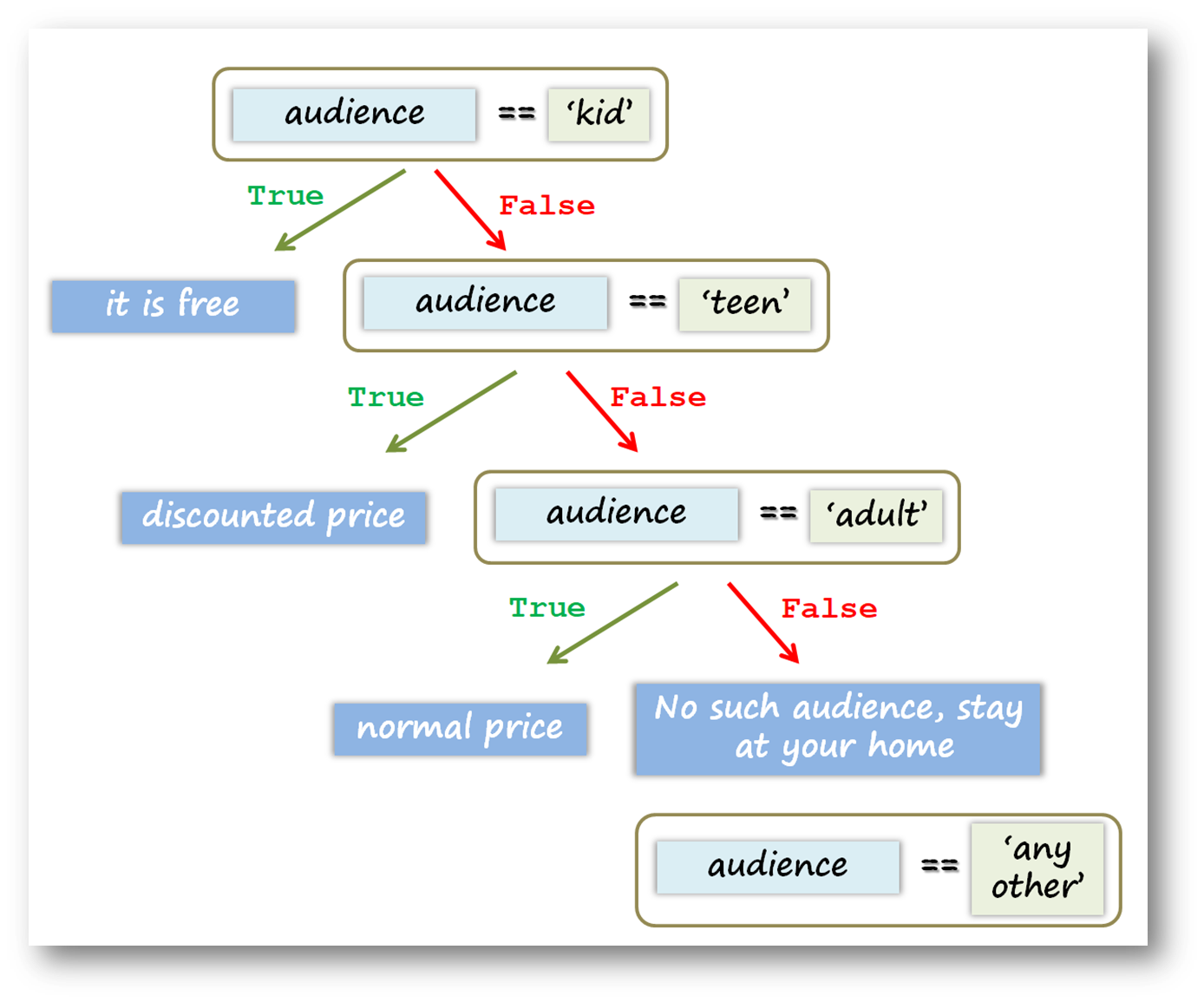


Diagram of Logic Flow

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) 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.

Lets 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 :



1

2

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).